Temporal Logic Encodings for SAT-based Bounded Model Checking

Daniel Sheridan

Doctor of Philosophy

Laboratory for Foundations of Computer Science

School of Informatics

University of Edinburgh

2005

Abstract

Since its introduction in 1999, bounded model checking (BMC) has quickly become a serious and indispensable tool for the formal verification of hardware designs and, more recently, software. By leveraging propositional satisfiability (SAT) solvers, BMC overcomes some of the shortcomings of more conventional model checking methods.

In model checking we automatically verify whether a state transition system (STS) describing a design has some property, commonly expressed in linear temporal logic (LTL). BMC is the restriction to only checking the looping and non-looping runs of the system that have bounded descriptions. The conventional BMC approach is to translate the STS runs and LTL formulae into propositional logic and then conjunctive normal form (CNF). This CNF expression is then checked by a SAT solver.

In this thesis we study the effect on the performance of BMC of changing the translation to propositional logic. One novelty is to use a normal form for LTL which originates in resolution theorem provers. We introduce the normal form conversion early on in the encoding process and examine the simplifications that it brings to the generation of propositional logic. We further enhance the encoding by specialising the normal form to take advantage of the types of runs peculiar to BMC. We also improve the conversion from propositional logic to CNF.

We investigate the behaviour of the new encodings by a series of detailed experimental comparisons using both hand-crafted and industrial benchmarks from a variety of sources. These reveal that the new normal form based encodings can reduce the solving time by a half in most cases, and up to an order of magnitude in some cases, the size of the improvement corresponding to the complexity of the LTL expression. We also compare our method to the popular automata-based methods for model checking and BMC.

Acknowledgements

I would like to thank to my supervisors at York, Toby Walsh and Alan Frisch, my supervisors at Edinburgh, Paul Jackson and Kousha Etessami, and my colleagues at IRST, Alessandro Cimatti and Marco Roveri, and Roberto Sebastiani at the University of Trento.

I am grateful for the support and encouragement of many people in the BMC community, especially of Ofer Strichman and Armin Biere, and in the SNF community, especially Michael Fisher and Clare Dixon. The support of the APES research group was also invaluable.

I want to thank my employers, Adelard LLP, for their tolerance and patience while I completed my thesis.

Finally, this thesis would never have been completed without the support and encouragement I received from my partner Joanne Allen.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Daniel Sheridan*)

Table of Contents

Abstract iii

Acknowledgements iv

Declaration v

Notation Conventions xvi

1 Introduction 1

1.1 Model Checking 1

1.2 Propositional Satisfiability 2

1.2.1 Encoding to SAT 4

1.3 Bounded Model Checking 4

1.4 Thesis Overview 5

1.4.1 Goal of the Work 6

1.4.2 Structure of the Thesis 7

1.4.3 Contributions 8

1.5 Related Work 9

1.5.1 Alternative BMC Encodings 9

1.5.2 Other Approaches 10

2 Background 15

2.1 Propositional Logic 16

2.1.1 Context Functions 18

2.2 Boolean Satisfiability 19

2.2.1 Conjunctive Normal Form 20

2.3 Boolean Formula Representation 30

2.3.1 Boolean Circuits 30

2.4 State-Transition Systems 35
2.4.1 Büchi Automata 35
2.4.2 Kripke Structures 36
2.5 Temporal Logic 37
2.5.1 LTL with Past Operators 41
2.5.2 Computational Tree Logic 43
2.5.3 Context Functions in LTL 43
2.5.4 Fixpoint Formulations of LTL 44
2.6 Model Checking 45
2.6.1 Symbolic Model Checking 46
2.6.2 Fairness 46
2.7 Summary 47

3 Bounded Model Checking **49**
3.1 Infinite and Finite Paths 49
3.1.1 *k*-Prefix Paths 50
3.1.2 *k*-Loop Paths 54
3.1.3 Combined BMC 57
3.1.4 Witness Length 59
3.2 The Bounded Model Checking Encoding 60
3.2.1 States and the Model 61
3.2.2 *k*-Prefix Paths 61
3.2.3 *k*-Loop Paths 62
3.2.4 Correctness 62
3.2.5 Encoding Fairness 64
3.3 Encoding Approaches from the Literature 65
3.4 Complexity of Encodings 66
3.5 Applications of BMC 67
3.6 Summary 69

4 The Separated Normal Form **71**
4.1 MetateM and SNF for PTL 71
4.1.1 Formal Definition of SNF 73
4.2 Recasting SNF using LTL 73
4.3 Denotational Semantics and QLTL 75
4.3.1 Quantified LTL 75

4.3.2 Denotational Semantics . . . 76
4.3.3 Context Functions . . . 79
4.3.4 Fixpoints . . . 80
4.4 Transformation to SNF . . . 80
4.4.1 Renaming . . . 81
4.4.2 Fixpoint Unwinding . . . 82
4.4.3 Fixpoint Characterisation . . . 82
4.4.4 Replacement of Duplicate Subformulae . . . 85
4.5 Conversion Strategies . . . 85
4.5.1 Top-down Conversion . . . 86
4.5.2 Bottom-up Conversion . . . 90
4.5.3 Discussion: the Propositional Form . . . 93
4.6 Summary . . . 96

5 Using the Separated Normal Form for BMC 103
5.1 Motivation . . . 103
5.2 SNF Formulae over Prefix and Loop Paths . . . 104
5.2.1 Denotational Semantics of Quantifier-free QLTL . . . 105
5.2.2 *k*-Prefix Paths . . . 105
5.2.3 *k*-Loop Paths . . . 108
5.3 SNF and BMC . . . 109
5.3.1 Encoding PSNF for BMC . . . 110
5.3.2 Improved Encodings . . . 115
5.3.3 A Linear-Space Encoding . . . 117
5.4 Further Optimisations . . . 119
5.4.1 SNF Variables . . . 120
5.4.2 **G** in the *k*-Prefix Path Case . . . 120
5.4.3 Variables and Eventualities . . . 122
5.5 Related Work . . . 123
5.6 Summary . . . 124
5.6.1 Transformation and Encoding . . . 125

6 CNF Conversion of RBCs 129
6.1 CNF Conversions on Linear Trees . . . 130
6.1.1 The Standard and Definitional Conversions . . . 131
6.1.2 Polarity-Dependant Renaming Conversions . . . 132

6.1.3 The Conversion due to Boy de la Tour 135
6.2 The Compact Conversion 141
6.3 Optimality of the Compact Conversion for Linear Trees 142
6.3.1 Children of Positive Polarity Conjunctions 145
6.3.2 Children of Negative Polarity Conjunctions 146
6.3.3 Both Polarities 147
6.3.4 Extension to RBCs 151
6.3.5 Implementation 152
6.4 Summary 153

7 Evaluation of SNF-style Encodings 155
7.1 Experimental Framework 155
7.1.1 Overview of NuSMV 155
7.1.2 Implementation 157
7.1.3 Platform 158
7.2 Hypotheses 158
7.3 Random Experiments 159
7.3.1 Models and Formulae 159
7.3.2 Clauses and Propositions 159
7.3.3 Solving Time 168
7.3.4 Encoding Time 172
7.4 Distributed Mutual Exclusion 174
7.5 Summary 175

8 SNF versus Automata Methods for BMC 177
8.1 Background: Automata and LTL Model Checking 177
8.2 Büchi Automata 179
8.2.1 Overview of GPVW 179
8.2.2 Improvements to GPVW 180
8.2.3 Büchi Automata Usage in Practice 184
8.2.4 Bounded Model Checking with Büchi Automata . . . 184
8.3 Alternating Automata 187
8.3.1 LTL to VWAA Conversion 188
8.3.2 BMC with Alternating Automata 192
8.4 Alternating Automata and SNF 195
8.5 Related Work 196

8.6 Summary and Conclusions 199

9 Conclusions **201**

9.1 Review of Thesis 201

9.2 Conclusions 202

10 Future Work **205**

10.1 Encoding the model 205

10.1.1 BDD to RBC conversion 205

10.1.2 Specialised Propositional Encodings 206

10.1.3 Encodings Inspired by Digital Electronics 210

10.2 Development of the Semantics 211

10.2.1 Maximality of Semantics 211

10.2.2 Exploiting the Complete Semantics 211

10.3 Automata-Inspired Approached 212

10.3.1 LTL Simplifications 212

10.3.2 Alternating Automata 212

10.3.3 Direct Application of the Fixpoint Transformations . . 212

A Publications **215**

A.1 "A Fixpoint Based Encoding for Bounded Model Checking" . 217

A.2 "Bounded Verification of Past LTL" 235

A.3 "Clause Form Conversions for Boolean Circuits" 251

A.4 "BMC with SNF, Alternating Automata and Büchi Automata" 267

Bibliography **285**

Index **296**

List of Figures

2.1 The semantics of propositional logic 17
2.2 The NNF conversion function $\textsc{Nnf}(f)$ 23
2.3 The standard CNF conversion function $\textsc{Cnf}(f)$ for $f \in \mathsf{nnf}$ . . 24
2.4 The definitional renaming function 26
2.5 The structure preserving renaming function 29
2.6 Example RBCs showing vertex labelling 32
2.7 The definitional clause form conversion for RBCs 34
2.8 The semantics of LTL 40
2.9 The NNF conversion function $\textsc{Nnf}(\phi)$ extended to LTL 41
2.10 The semantics of the PLTL past time operators 43

3.1 Three finite cases for $\mathbf{F}\,\phi$ 53
3.2 The sound semantics of NNF LTL for k-prefix paths 54
3.3 The complete semantics of NNF LTL for k-prefix paths 55
3.4 Graphical depiction of k, l, and p 56
3.5 The projected semantics of LTL for k-loop paths 58
3.6 Path examples 59
3.7 Counter example, $n = 3$ 60

4.1 Denotational semantics of QLTL 78
4.2 Fixpoint characterisations of LTL operators with $\mathbf{X}$ 84
4.3 Top-down implementation of PSNF conversion 95

5.1 The sound semantics of QLTL for k-prefix paths 106
5.2 The projected semantics of QLTL for k-loop paths 109
5.3 Illustration of loop lengths in a 5-1-loop path 121
5.4 The SNF transformation functions for BMC 127
4.4 Example conversion of $a\,\mathbf{U}(\mathbf{G}\,b)$ to SNF 98

4.5 Example conversion of $a\,\mathbf{U}(\mathbf{G}\,b)$ to SNF 100

6.1 The standard clause form conversion for tree RBCs 133
6.2 The structure-preserving renaming construction SP(T) 134
6.3 Example: structure-preserving CNF applied to an RBC 135
6.4 The vertex-based renaming construction ren($T, \mathbf{R}$) 136
6.5 Renaming sets construction for the Boy de la Tour conversion 140
6.6 Renaming sets construction for the compact conversion 142
6.7 RBC subgraphs for the optimality proofs 146
6.8 RBC subgraphs for the equivalence discussion 151

7.1 $|\phi|$ versus clauses $k = 15$ 161
7.2 $|\phi|$ versus clauses $k = 30$ 161
7.3 $|\phi|$ versus propositions $k = 15$ 162
7.4 $|\phi|$ versus propositions $k = 30$ 162
7.5 $|\phi|$ versus clauses $k = 30$ (compact CNF conversion) 163
7.6 $|\phi|$ versus propositions $k = 30$ (compact CNF conversion) . . . 163
7.7 k versus clauses $|\phi| = 3$ 165
7.8 k versus propositions $|\phi| = 3$ 165
7.9 k versus clauses $|\phi| = 16$ 166
7.10 k versus propositions $|\phi| = 16$ 166
7.11 k versus clauses $|\phi| = 16$ (compact conversion) 167
7.12 k versus propositions $|\phi| = 16$ (compact conversion) 167
7.13 $|\phi|$ versus solving time in zChaff at $k = 30$ 169
7.14 $|\phi|$ versus solving time in BerkMin at $k = 30$ 169
7.15 k versus solving time in zChaff at $|\phi| = 16$ 170
7.16 k versus solving time in BerkMin at $|\phi| = 16$ 170
7.17 k versus solving time in zChaff at $|\phi| = 16$ (compact conversion) 171
7.18 k versus solving time in BerkMin at $|\phi| = 16$ (compact conversion) 171
7.19 $|\phi|$ versus encoding time at $k = 30$ 173
7.20 k versus encoding time at $|\phi| = 16$ 173

8.1 Example alternating automaton and part of a run 189

10.1 Examples of BDD transformations for XOR 207

List of Tables

3.1 The BMC encoding for LTL, prefix path case 62
3.2 The BMC encoding for LTL, loop path case 63
3.3 The factorised BMC Encoding for LTL 67

5.1 The BMC encoding for PSNF rules, prefix path case 111
5.2 The BMC encoding for PSNF rules, loop path case 113
5.3 The BMC encoding for PSNF rules 127

6.1 The clause counting functions $p^+(V)$ and $p^-(V)$ 137
6.2 Computation of the coefficients a_V^T and b_V^T 138
6.3 The renaming-compensated clause counting functions 139

7.1 Encodings to be evaluated 158
7.2 Distributed mutual exclusion benchmark results 176

8.1 Comparison of automata and SNF techniques for BMC 197
8.2 Further results 198

Notation Conventions

Typographical Conventions

We use certain typographical features to help distinguish between the types of variables used in the thesis.

Convention	Meaning	Example	Reference
italic lower	propositional formulae	f	Section 2.1
italic upper	graphs, vertices or edges	E	Section 2.3.1
roman upper	conversion functions	$\textsc{Cnf}$	Section 2.2.1
bold upper	sets of graphs, vertices or edges	$\mathbf{V}$	Section 2.3.1
greek	temporal formulae, paths	ϕ, ψ, π	Section 2.5
greek upper	context functions	Φ, Ψ	Section 2.1.1
sans-serif	languages	prop, ltl	Section 2.1

Unusual Notation

In some cases we make a break from convention in order to help clarify meanings or emphasise distinctions.

Symbol	Meaning	Reference
$\rightarrow$	Implication $a \rightarrow b \equiv \neg a \vee b$	Section 2.1
$\Rightarrow$	Implication in SNF rules	Section 4.1.1
$[\![\phi]\!]$	Propositional encoding of temporal ϕ	Section 3.2
$\langle\!\langle \phi \rangle\!\rangle$	Denotational semantics of ϕ	Section 4.3

Symbolic Conventions

By convention, some symbols are always used with the same meaning. This enables us to remove a large number of explanations and side conditions.

Symbol	Meaning	Reference
$\top$	The true proposition $\top \equiv a \vee \neg a$	Section 2.1
$\bot$	The false proposition $\bot \equiv \neg\top$	Section 2.1
a	Atomic proposition $a \in AP$	Section 2.1
α	Set of atomic propositions $\alpha \subseteq AP$	Chapter 8
$\hat{\alpha}$	Set of sets of atomic propositions $\alpha \subseteq 2^{AP}$	Chapter 8
$f, g, f_0, g_0, \ldots$	Propositional formula	Section 2.1
$\phi, \psi, \phi_0, \psi_0, \ldots$	Temporal formula	Section 2.5
α	Temporal variable $\alpha \in Q$	Section 4.3.1
q	Denotational variable $q \subseteq \mathbb{N}$	Section 4.3.2
r_f, r_ϕ	Proposition or variable renaming f or ϕ	Section 2.2.1
π	Infinite path	Section 2.5
ϖ	Finite path	Section 3.1
ρ	Environment	Section 4.3.2
ϱ	Bounded environment	Section 5.2.1
l	Loopback index	Section 3.1.2
k	Bound	Section 3.1
$\emptyset$	The empty set	

Chapter 1

Introduction

Bounded model checking brings together the formal hardware verification approach of symbolic model checking and the generalised theoretical approach of propositional satisfiability solving. We suggest that the efficiency of the bounded model checking procedure can be improved by changes to the encoding from the model checking domain to the propositional logic domain.

1.1 Model Checking

Model checking [25] is a formal hardware verification technique: a method for showing that a model of a hardware design satisfies, or violates, specifications written to capture its intended behaviours. We begin with a model in terms of a state transition system (STS): either explicitly represented or given symbolically in some restricted programming language; and the specification in a temporal logic: a logic including operations for the quantification of time. Solving a model checking problem involves finding an example of error behaviour, or an assertion that no such trace exists.

A number of techniques exist for showing that an STS fulfils the requirements of a specification. The earliest approaches were *explicit-state* algorithms. In such algorithms, sets of states are represented by explicitly enumerating the individual states in each set. That is, intermediate results of calculations such as the set of reachable states of the system are represented explicitly. These algorithms can be highly efficient (consider, for example, SPIN [55], which uses partial order reductions to represent equivalence classes of states). Although they can handle STSs with several million states they suffer from the large

size of the state spaces representing hardware designs of industrial complexity (since the number of states is exponential in the number registers in the circuit).

Symbolic model checking [21, 32] is seen by many as the breakthrough that made model checking as widely applicable as it is today. The key idea here is to represent sets of states symbolically using, for example, a Boolean function over the states, rather than simply listing the set membership. This is easily obtained from a symbolic representation of the transition relation: the designer is likely to describe the model in appropriate terms ("the next value of x is $x + 1$ unless *stop* is true…") rather than explicitly ("from state 39 take a transition to state 40 if *stop* is true…"). Central to practical symbolic model checking is the use of the data structure *binary decision diagrams* (BDDs) [20], which represent an efficient way of storing a function from a set of variables to a truth value; for symbolic model checking this is the function from variables representing the 'current' state to the truth value corresponding to whether the described state is reachable. A restriction on BDDs, *reduced, ordered* BDDs, have the useful property of canonicity: if two function give the same result at every point, their representations are the same. This means that temporal properties may be unfolded using the calculus of fixpoints, which involves detecting whether a transformation of a function changes it.

This technique made symbolic model checking applicable to systems of over 10^{20} states, and various further refinements on the construction and handling of BDDs as well as techniques for abstraction of the model have been employed to push this limit still higher. Symbolic model checking is sufficiently powerful to be applied to industrial designs with companies such as Intel and IBM as part of their internal verification processes.

While BDDs represent most functions compactly, there are some notable exceptions, such as multiplier circuits. The so-called "state-explosion" problem is a significant one considering that a major application domains for model checking is the microprocessor.

1.2 Propositional Satisfiability

Propositional satisfiability (SAT) is the quintessential NP-complete problem. The problem is as follows: given a set of Boolean variables and a propositional formula constraining them, to find an assignment of truth values to variables

such that the formula evaluates to true. Informally, the time taken to check a variable assignment is polynomial (in fact, linear) in the size of the formula, but as there are an exponential number of different assignments to the variables, only a non-deterministic procedure can guarantee to exhibit polynomial performance. The issue of whether P=NP—whether a polynomial time SAT algorithm exists—is one of the Clay Mathematics Institute *Millennium Problems* [29], and carries a million-dollar prize.

Despite this algorithmic complexity, the SAT problem has been extensively studied. By limiting the formula to clause form (a conjunction of disjunctions of variables or their negations) it is possible to find easily solved subsets of the problem. For example, if every clause consists of just two literals (2-SAT), the problem may be solved in linear time using the algorithm of Aspvall, Plass, and Tarjan [2]. For clauses of three literals (3-SAT), current solving procedures exhibit a phase transition in the fraction of random problems solvable as the clause/variable ratio is increased (Gent and Walsh [54]). The transition region corresponds to the most difficult problems: problems in this region are observed to take exponential time to solve, decaying to linear time further from the transition.

Unlike random problems, SAT problems that occur in practice turn out to be surprisingly amenable to machine solving. The Davis-Putnam algorithm [34], published in the 1960s as a paper-based resolution system, was transformed into a practical algorithm by Davis, Logemann, and Loveland [35], trading exponential space for exponential time to form the DPLL procedure. This depth-first backtracking search of the assignment space has seen numerous developments to improve its efficiency, such as non-linear backtracking directed by conflicting variable assignments [86] and the incorporation of clauses removing discovered conflicts from the search space (conflict learning) [103]; and numerous implementation developments including novel data structures [102] and manipulation methods [79].

The quality of available solvers for SAT has been driven up by the SAT competition held annually at the SAT conference. Most recently, 55 solvers were tested on 999 benchmark problems from industry and academia, including model checking, equivalence testing, planning, as well as random problems [9].

1.2.1 Encoding to SAT

Non-deterministic polynomial (NP) problems are solvable in polynomial time given a perfect oracle (equivalently, answers are testable in polynomial time). Problems in the complexity class NP-complete may be converted in polynomial time to an instance of any other problem. NP-complete includes such practical and interesting problems as certain restrictions of planning [61] and timetabling [30], as well as logic games such as Tetris[1] [19] and Minesweeper [69]. Garey and Johnson [52] list a series of further problems, and a more up-to-date list (88 problems at the time of writing) is maintained by Dunne [41]. The availability of SAT solvers make SAT a particularly inviting target for solving a variety of NP-complete problems.

The first example of a successful SAT encoding from another problem domain was the problem of planning [75]. Despite the early state of development of SAT solvers at the time, the first SAT-based planner, SATPLAN by Kautz and Selman [67] was at the cutting edge when it was released. A later implementation improving the encoding and designed to take advantage of the wide variety of competing SAT solvers, Blackbox [68], became the leading planner two years later (a domain specific solver was briefly the winner). Blackbox is a key example of the advantage of using generic SAT procedures: although the encoding has changed very little in the last 10 years, the SAT solvers have changed substantially, and Blackbox remains the leading planner. Blackbox has leveraged the improvement in SAT solving in much the same way that improving an implementation of Java, or the design of a microprocessor, simultaneously improves all applications that run on it.

1.3 Bounded Model Checking

Bounded model checking (BMC) [11, 12] was introduced as a solution to the state-explosion problem, avoiding BDDs altogether by taking the same route as planning research—encoding to SAT. Unlike planning, however, model checking of the linear-time logic chosen (LTL) is PSPACE-complete, implying an exponential increase in problem size during encoding.

To avoid this obvious drawback, a bound is placed on the number of

[1] Tetris is a registered trademark of The Tetris Company, LLC.

transitions considered. By observing whether the transitions form a loop in the STS, even infinite-time properties may be verified by this method. The drawback is that error traces beyond the bound are not considered. BMC is therefore only complete if a sufficiently high bound (the *diameter* of the STS) is chosen—an NP-complete problem in itself. In practice, classical BMC is typically used to provide an assurance of bounded correctness, and we attempt to find error traces at increasing bounds until one is found or the desired level of assurance is reached. More advanced techniques exist for completing BMC based on induction or on techniques more closely related to BDD-based model checking. Experiments show that BMC is indeed able to solve cases that BDD-based model checking fails on, although the converse is also true: there is no clear winner between the two competing techniques.

The classical bounded model checking encoding is as follows. We begin with enough propositional instances of the state variables to cover the bounded number of states. A series of constraints are placed on the variables to ensure that they can only take on the values of a valid path through the STS. Since the LTL specification is expected to hold for every valid path in the system, it is negated and encoded as further constraints on the state variables: they may now take on only values corresponding to an error trace. The resulting system of propositional variables and constraints forms a SAT problem, satisfiable only when an error trace exists within the bound.

The problem of encoding is therefore to find appropriate constraints to express the STS and specification succinctly. The former is very similar to the problem of representing an STS as a BDD: the transition function is already made available by the designer. Encoding the specification presents a significant challenge; the original presentation includes an encoding based on a direct conversion of the semantics of temporal logic to propositional logic, and suffers from a complexity which increases rapidly with the complexity of the specification itself.

1.4 Thesis Overview

In order to reduce the scope of the thesis to manageable proportions, we make the following restrictions on the work:

- We focus only on classical BMC (the procedure suggested by Biere et al.

[12] and its extensions) as the model checking–via–SAT methodology. We regard other developments in the field as not integral, even though the techniques presented here may be more widely applicable.

- We ignore the encoding of the model to propositional logic. The similarity between this problem and the construction of BDDs suggests that the issue has been extensively explored elsewhere. In fact, in the research platform chosen, the model is encoded by first constructing a BDD representation and then converting it to propositional logic.

- We treat SAT technology as a black-box. That is, the investigation of the encoding methods focuses on the size of the encoding, the time to generate, and the time to solve, but ignores the details of the relationship between the encoding and the internal behaviour of a particular SAT solver.

- We focus on clause form–based SAT technology. At the time of writing, non-clausal SAT solvers are emerging as potential competitors to the clause-form solvers but the results are not sufficiently conclusive to justify a wholesale switch.

- For the performance comparisons we examine only a handful of the leading SAT solvers, assuming that their exceptional performance in the SAT solver competition is indicative of their performance on the new encodings.

1.4.1 Goal of the Work

We wish to understand the effect on the performance of BMC of changing the encoding to propositional logic. In particular, we are interested in two parts of the overall encoding process: encoding the specification to propositional logic, and converting the resulting propositional formula to CNF form, suitable for input to the SAT solver.

For the specification, we take our inspiration by looking at other fields where temporal logic is used. In particular, in the field of resolution-based theorem proving for temporal logic, the use of a temporal logic normal form is well established. The nature of the normal form is such that the burden

of encoding to propositional logic is significantly reduced by performing the conversion to it; our hypothesis is that this also translates into a reduction in the time taken by the SAT solver on a given problem. We also investigate the implications that the bounded nature of BMC have on the generation of the normal form, and by returning to the roots of the normal form, we try to further improve its utility to BMC.

The conversion from general propositional logic to clause form has been extensively studied for first-order theorem proving, but has been largely ignored in the new field of propositional satisfiability. We take the best performing conversion from the first-order domain and apply it to BMC problems producing a reduction in the size of problems passed on to the SAT solver. There is no reason, however, to expect a reduction in the solving time to be a result of a reduction in the problem size. Indeed, there is plenty of evidence for the introduction of *additional* clauses to an existing problem to reduce solving time. Our hypothesis here is that the number and type of the extra clauses introduced by the existing conversion method do not have this effect: that the reduction in problem size here is uniformly beneficial.

We investigate the hypotheses above by implementing the algorithms as part of the model checker NuSMV, and performing a series of detailed experimental comparisons. The benchmarks come from a variety of sources, including hand-crafted ones to demonstrate specific points, and publicly-available industrial benchmarks.

For a more theoretical perspective, and to further understand the normal-form-based encodings of the specification, we look at the standard technique for model checking linear-time logics used both in symbolic and explicit-state model checking: the conversion to automata.

1.4.2 Structure of the Thesis

As the work in this thesis draws from a variety of different fields, we begin by recapping the underlying theoretical material (Chapter 2). This places us in a suitable position to give a detailed discussion of classical bounded model checking and the existing BMC literature (Chapter 3).

As the core work in this thesis is on the normal-form-based specification encoding, we present the background of the normal form in Chapter 4. This

allows us to move on to the direct use of the normal form in BMC, and the scope for extending it to take advantage of the nature of BMC itself (Chapter 5). To consolidate this part of the thesis, we present an analysis from the point of view of automata-based linear-time model checking (Chapter 8).

The second major contribution of the work, the clause-form conversion, is described in Chapter 6. We give a detailed experimental analysis of all of the methods described in Chapter 7 before drawing our conclusions from the work and suggesting future expansion to it (Chapters 9 and 10).

1.4.3 Contributions

The contributions made by this thesis to the field are summarised here.

At the highest level, two significant methods for improving the performance of the BMC procedure given by Biere, Cimatti, Clarke, and Zhu [12] are presented. Firstly, an encoding for LTL to propositional logic is presented (Chapter 5) which grows linearly with the size of the input formula rather than polynomially.

This encoding grows out of a new and more thorough presentation (Chapter 4) of the conversion from LTL to the normal form SNF and its correctness than has previously been seen (e.g., in the work of Bolotov [14], Dixon [40], Fisher [46] and others). A new transformation procedure combining the advantages of bottom-up and top-down conversion is given. SNF has not previously been applied to a temporal logic over finite paths, and it has not previously been used for model checking.

The new presentation of the SNF encoding in Chapter 5 is a significant improvement in rigour and clarity over the previously published presentations (Cimatti et al. [24], Frisch et al. [48]). The relationship between SNF and alternating automata (Chapter 8) helps place the SNF in context with regards to other LTL model checking approaches.

In order to place this work in its proper setting, a new presentation of the BMC encoding is given in Chapter 3 which is intended to make the link between the infinite and bounded semantics. This results in a clearer and more thorough explanation of BMC than that available in the standard papers by Biere et al. [11, 12].

The second significant improvement in performance of BMC comes from

the conversion of propositional logic to clause form in the context of a Boolean circuit representation. The new procedure presented in Chapter 6 is a linear time algorithm which produces much more compact clausal representations of formulae. It is faster and considerably simpler than the other known optimal algorithms (Boy de la Tour [18] and Nonnengart, Rock, and Weidenbach [81]).

1.5 Related Work

As this material in this thesis brings together several related fields, each major chapter has its own discussion of related work. In this section, work is reviewed which is less closely related.

Approaches to obtaining the variations on the standard BMC encoding are discussed in Section 3.3 and the application of the BMC procedure to particular problems is discussed in Section 3.5. There is a small but significant body of work which focuses on other approaches to the improvement of BMC including changes in the SAT solver, the way the SAT solver is used, and also the use of induction in BMC. Encodings and other model checking–via–SAT techniques not directly related to the standard Biere et al. encoding are discussed below.

The main papers related to SNF and its application to various temporal logics are described in Section 4.1. The relationship between SNF and automata methods for model checking LTL is explored in Chapter 8.

Although the use of SNF for model checking and the projection of SNF to bounded temporal logic is entirely new work, there is a growing body of work concerning an alternative linear space BMC encoding. This is discussed in Section 5.5.

The chapter on CNF conversions describes the principle existing conversion techniques in Sections 6.1.1, 6.1.2 and 6.1.3, recasting them in the same framework as the new work.

Finally, related publications written by the author of this thesis are given in Appendix A.

1.5.1 Alternative BMC Encodings

There are three papers of which we are aware that consider specification encodings for LTL which are different from that given by Biere et al. [12].

de Moura, Rueß, and Sorea [36] suggest the use of LTL to Büchi automata translations, already established as the method for handling LTL in both explicit-state and symbolic model checking, as a way of sidestepping the encoding issue. This work is supported by theoretical observations from Clarke et al. [28] who are able to give a smaller upper bound on the complexity of BMC using this encoding, compared to that of Biere et al.

1.5.2 Other Approaches

There are several alternative approaches to the broad goal of speeding up the solving of bounded model checking. As they are orthogonal to the approach taken in this thesis (indeed, preliminary research suggests that some of these techniques can be fruitfully combined with those described here) we summarise them only briefly.

1.5.2.1 Specialised SAT Solvers

An alternative to changing the BMC encoding as proposed here is to change the SAT solver to better deal with the type of propositional logic generated by BMC. This has been examined in detail by Strichman [92]. He investigates four key methods for improving the SAT solver GRASP [73] (listed below) and is able to demonstrate a significant reduction in the time taken on several benchmark problems. It is worth noting that GRASP is no longer considered state-of-the-art, and it is not clear whether Strichman's techniques would have the same impact if applied to a more modern solver.

Constraints replication The formulae generated by BMC are very repetitive: the same formula is repeated multiple times over different variables. When a property of that formula is deduced by the SAT solver, such as a conflict assignment, it may be possible to project that property over all instances of the formula. Conflict learning can cause a dramatic reduction in the search space, and this technique generalises this result.

Variable ordering The order in which variables are chosen for assignment during search can have a significant effect on the time taken to find assignments. Strichman suggests techniques for determining the variables which have the greatest impact on the search space, in terms of

the number of other variables whose value depends on them. This static ordering is in contrast to the dynamically adjusted variable ordering usually used.

Assignment to the chosen variable Strichman explores various strategies for choosing the value assigned to the chosen variable during search. One suggested method is to assume that in successive states in the transition system, variables remain constant, and to base initial assignments to variables on the corresponding variables in earlier states.

Splitting on state variables As a complement to the specialised variable ordering, Strichman follows Giunchiglia and Sebastiani [58] in prioritising those variables which occur in the model over those which stem from the CNF conversion (see Section 2.2.1.1). Giunchiglia and Sebastiani suggest this for defining a non-clausal SAT solver; we believe that the success of this approach supports the argument that the quality of the CNF conversion (as explored in Chapter 6) has a significant impact on the overall performance of the procedure.

1.5.2.2 Incremental BMC

BMC by its nature involves placing an arbitrary bound on the number of future states considered. While theoretical results exist for finding (with various degrees of tightness) the bound required for a given model and a given specification, these can be prohibitively hard to determine. A common methodology for finding a bug in a model is therefore to attempt to find a bug at an initial bound k_0, and if none is found, to repeat the process at $k_0 + 1$, $k_0 + 2$, ..., until either a bug is found or some chosen maximum is reached.

Naturally, such an approach involves a significant amount of repeated work since a new SAT problem is created and solved at each iteration. Strichman [95] proposes a technique for overcoming this issue for the special case of invariant problems (specifications of the form $\mathbf{G}\, f$). At each iteration, the set of learned conflict clauses is gathered from the SAT solver, generalised as described above, and then included directly in the successive iterations. This means that a significant proportion of the search space, pruned by earlier iterations, is removed from the later iterations.

Benedetti and Bernardini [7] takes Strichman's suggestion one step further and integrates the SAT solver directly with the BMC encoding procedure. At the end of each iteration, the SAT solver is 'paused' and the new clauses for the next iteration are added to solver's internal state. This means that all of the learning and search space elimination is carried across directly to the following iteration. In addition, the startup cost of the SAT solver is avoided. Benedetti and Bernardini give a procedure for the full range of specifications, unlike the Strichman procedure. The disadvantage of this approach is that it breaks one of the most interesting and useful properties of encoding to SAT: the interchangeability of solvers. No standard library interface has yet been agreed upon to complement the DIMACS file format [38] implemented by all solvers.

The work of Latvala et al. [71] (see Section 5.5) has also been extended to an incremental algorithm (Heljanko, Junttila, and Latvala [62]) using a similar approach to that of Benedetti and Bernardini.

1.5.2.3 Induction

In a paper published the same year as the first BMC paper, Biere, Cimatti, Clarke, Fujita, and Zhu [10] describe a more efficient method of checking that invariants hold by following the form of mathematical induction. In their scheme, an invariant is shown to hold by checking firstly that it holds in the initial states of the transition system, then by checking the inductive property that, if the property holds in a given state, it also holds in the successor states. This is much simpler property to prove than for the general BMC approach, and is also an unbounded method as it checks every transition and hence every state.

Sheeran, Singh, and Stålmarck [88] point out that the inductive approach of Biere et al. is only guaranteed to be complete if every state in the transition system is reachable from one of the initial states. If this is not the case then an unreachable state in which the invariant does not hold can cause the induction to fail—even though the property holds in practice. Sheeran et al. propose a more complex scheme which extends the inductive hypothesis over multiple states. They consider the inductive hypothesis to be met only if the run of states can be related back to the initial states.

McMillan [78] takes a very different approach to BMC by using SAT to implement a propositional formula simplification to replace the use BDDs

in symbolic model checking. The paper makes the observation that conflict clauses are unsatisfiable precisely when the original formula is invalid, and so the complete set of conflict clauses is equivalent to the original formula. The algorithm collects conflict clauses as they are generated, and each time a satisfying assignment is reached it is negated and used to refine the input formula. As the result is a conjunction of conflict clauses, it is already in CNF, making this algorithm ideal for use in an iterative procedure.

In this framework, it is also simple to remove universally quantified variables: it is equivalent to remove them from the conflict clauses as they are generated. The intended application of this algorithm — to find the characteristic formulae of the sets of states in which all successor states satisfy a property — relies on this universal quantification.

A further paper by McMillan [76] describes an approach more similar to that of Sheeran et al. [88]. This paper suggests that Craig interpolants are used during the computation of reachability to compute an overapproximation of the set of states at successive distances from the initial set of states. The idea is to make use of the refutation proof produced by some SAT solvers when returning an 'undatisfiable' result. Eventually, this approach reaches a fixpoint expression for the set of reachable states, and can thus terminate with a success or failure result depending on whether this expression and the required property are mutually satisfiable. McMillan is able to demonstrate a practical implementation of a complete model checking algorithm which outperforms BDD-based model checkers on many problems, and is roughly comparable with (incomplete) BMC approaches.

Chapter 2

Background

The work presented in this thesis draws from a number of different fields, so there is a wide variety of background material to introduce. In this chapter we cover the formal framework and background material required to support the introduction of bounded model checking in the following chapter. Further background material specific to each chapter will be introduced alongside the new material.

The order in which the material is presented represents the dependencies between the fields. Model checking, fully defined in Section 2.6, is the process of showing that the behaviour of a design or a problem behaves according to some property, usually corresponding to an aspect of its intended behaviour. The model is typically given as some type of automaton (Section 2.4) while the property is given in a temporal logic (Section 2.5). Bounded model checking (Chapter 3) is a modern approach to solving model checking problems by exploiting propositional SAT solver technology (Section 2.2). This raises two issues which are necessary for exploiting the technique: representing propositional formulae (Section 2.3) and converting them to a suitable input format (Sections 2.2.1 and 2.3.1.2) before solving.

The implementation work which forms the basis of the experimental evaluation in Chapter 7 is an enhancement of the model checker NuSMV [22], so much of the background material is oriented towards implementation decisions made by the authors of NuSMV. The relevant sections highlight where competing implementations have taken radically different routes.

2.1 Propositional Logic

We briefly discuss the syntax and semantics of conventional propositional logic as it is a key building block for the other logics used in this thesis.

Let AP be the set of all atomic propositions (sometimes called *propositional variables*), and $\top$ and $\bot$ correspond syntactically and semantically to the true and false propositions respectively.

Definition 2.1.1 (propositional logic formula)
The set prop of propositional logic formulae over AP is defined as the smallest set obtained from the productions below.

$$\frac{}{\top, \bot \in \mathsf{prop}} \quad \frac{a \in AP}{a \in \mathsf{prop}} \quad \frac{f \in \mathsf{prop}}{\neg f \in \mathsf{prop}} \quad \frac{f_0, \ldots, f_n \in \mathsf{prop},\ n \geq 1}{f_0 \wedge \cdots \wedge f_n \in \mathsf{prop}}$$

$$\frac{f_0, \ldots, f_n \in \mathsf{prop},\ n \geq 1}{f_0 \vee \cdots \vee f_n \in \mathsf{prop}} \quad \frac{f_0, f_1 \in \mathsf{prop}}{f_0 \rightarrow f_1 \in \mathsf{prop}} \quad \frac{f_0, f_1 \in \mathsf{prop}}{f_0 \leftrightarrow f_1 \in \mathsf{prop}}$$

Although we introduce $\wedge$ and $\vee$ and n-ary connectives above, we allow them to be read as combinations of binary instances of the connectives. The usual rules for distribution and association (listed below) allow us to pick any convenient bracketing in order to do this. We make use of this property when defining transformations over propositional logic formulae as it allows us to give the transformations in simpler terms.

The set of atomic propositions in a propositional formula is obtained by the function $ap : \mathsf{prop} \rightarrow 2^{AP}$: the set of atomic propositions in f is written $ap(f)$. We give truth-value assignments to the atomic propositions in f using a partial function $\sigma : AP \nrightarrow \{\top, \bot\}$. The semantics of propositional logic are defined in terms of the truth-value assignment.

Definition 2.1.2 (semantics of propositional logic)
The semantic interpretation of a propositional formula f with respect to an assignment function which gives a value to every atomic proposition occurring in f, that is, $\sigma : AP \nrightarrow \{\top, \bot\}$ where $dom(\sigma) \supseteq ap(f)$, is given in Figure 2.1.

Definition 2.1.3 (extension to partial assignments)
The semantics given Definition 2.1.2 is extended to the case where σ does not give an assignment to every atomic proposition in the propositional

$$
\begin{array}{lcl}
\sigma \models \top & & \\
\sigma \not\models \bot & & \\
\sigma \models v & \Leftrightarrow & \sigma(v) \quad \text{for } v \in AP \\
\sigma \models \neg f & \Leftrightarrow & \neg(\sigma \models f) \\
\sigma \models f_0 \wedge \cdots \wedge f_n & \Leftrightarrow & \sigma \models f_0 \text{ and } \ldots \text{ and } \sigma \models f_n \\
\sigma \models f_0 \vee \cdots \vee f_n & \Leftrightarrow & \sigma \models f_0 \text{ or } \ldots \text{ or } \sigma \models f_n \\
\sigma \models f_0 \rightarrow f_1 & \Leftrightarrow & \text{if } \sigma \models f_0 \text{ then } \sigma \models f_1 \\
\sigma \models f_0 \leftrightarrow f_1 & \Leftrightarrow & \sigma \models f_0 \text{ if and only if } \sigma \models f_1
\end{array}
$$

Figure 2.1: The semantics of propositional logic

formula f. Then $\sigma \models f$ if and only if for every extension of σ to the remaining atomic propositions, σ', it holds that $\sigma' \models f$. That is,

$$\sigma \models f \Leftrightarrow \forall \sigma' \supseteq \sigma \,.\, dom(\sigma') \supseteq ap(f) \rightarrow \sigma' \models f$$

Definition 2.1.4 (equivalence of propositional formulae)
Two formulae $f_0, f_1 \in \mathsf{prop}$ are *semantically equivalent* if they describe the same mathematical formula. That is, for every $\sigma \in AP \nrightarrow \{\top, \bot\}$, we have that $\sigma \models f_0$ if and only if $\sigma \models f_1$; we write $f_0 \equiv f_1$ in this case.

We say that f_0 and f_1 are *syntactically equivalent* if they have precisely the same syntactic structure; we write $f_0 = f_1$ in this case.

Clearly $f_1 = f_2$ implies $f_1 \equiv f_2$; the latter relation is a congruence, permitting the replacement of f_1 by f_2 without changing the meaning of a formula.

We list below, without proof, the distributive and associative rules and other useful identities of propositional logic.

$$
\begin{aligned}
f_0 \wedge f_1 \wedge \cdots \wedge f_n &\equiv (f_0 \wedge f_1) \wedge \cdots \wedge f_n \\
f_0 \vee f_1 \vee \cdots \vee f_n &\equiv (f_0 \vee f_1) \vee \cdots \vee f_n \\
(f_0 \wedge f_1) \vee f_2 &\equiv (f_0 \vee f_2) \wedge (f_1 \vee f_2) \\
(f_0 \vee f_1) \wedge f_2 &\equiv (f_0 \wedge f_2) \vee (f_1 \wedge f_2) \\
f_0 \leftrightarrow f_1 &\equiv (f_0 \rightarrow f_1) \wedge (f_1 \rightarrow f_0)
\end{aligned}
$$

$$(f_0 \wedge f_1) \wedge f_2 \equiv f_0 \wedge (f_1 \wedge f_2) \qquad (f_0 \vee f_1) \vee f_2 \equiv f_0 \vee (f_1 \vee f_2)$$
$$f_0 \wedge f_1 \equiv f_1 \wedge f_0 \qquad f_0 \vee f_1 \equiv f_1 \vee f_0$$
$$f_0 \wedge f_0 \equiv f_0 \qquad f_0 \vee f_0 \equiv f_0$$
$$f_0 \rightarrow f_1 \equiv \neg f_1 \rightarrow \neg f_0 \qquad f_0 \rightarrow f_1 \equiv \neg f_0 \vee f_1$$

2.1.1 Context Functions

Context functions provide a general mechanism for identifying and replacing subformulae. They are defined syntactically, rather than semantically, in terms of substitutions. To allow for the full expressivity of context functions (which we will need in Chapters 4 and 6), we define substitution as the replacement of an atomic proposition by a new subformula.

Definition 2.1.5 (substitution)
A substitution $f[g/a]$ is the replacement of every occurrence of a in f by g. Formally,

$$\begin{aligned}
a[g/a] &= g \\
a_0[g/a] &= a_0 && a_0 \in AP,\ a_0 \neq a \\
(\neg f_0)[g/a] &= \neg(f_0[g/a]) \\
(f_0 \wedge f_1)[g/a] &= (f_0[g/a]) \wedge (f_1[g/a]) \\
(f_0 \vee f_1)[g/a] &= (f_0[g/a]) \vee (f_1[g/a]) \\
(f_0 \rightarrow f_1)[g/a] &= (f_0[g/a]) \rightarrow (f_1[g/a]) \\
(f_0 \leftrightarrow f_1)[g/a] &= (f_0[g/a]) \leftrightarrow (f_1[g/a])
\end{aligned}$$

Definition 2.1.6 (context function)
A *context function* $F[\_]$ is a formula in propositional logic (Definition 2.1.1) extended with the atomic proposition $\_$ (*holes*). An instantiation $F[f]$ denotes the replacement of every hole by the formula f and may be seen as a short-hand for $(F[\_])[\phi/\_]$.

We define a context function by using syntactic pattern-matching of a propositional formula against an instantiated context function. That is, for propositional formula f with zero or more occurrences of subformula f_0, the assertion

$$F[f_0] = f$$

defines F to be context function with a hole for every occurrence of f_0 in f. Note that in this sense the definition is always assumed to be maximal: it always defines the context function with the highest number of holes.

2.2 Boolean Satisfiability

We say that a formula in propositional logic is *satisfiable* if there is an assignment to a subset of the atomic propositions of the formula such that the formula becomes equivalent to true. Software programs for finding such assignments, or for demonstrating that they do not exist, are called *SAT solvers*.

The Boolean satisfiability problem is probably the most well known NP-complete problem—informally, the number of assignments to the atomic propositions grows exponentially with the number of atomic propositions, while a given assignment may be verified to satisfy a formula in time proportional to the size of the formula. SAT has received a lot of attention in the last decade as many useful NP-complete problems can be more efficiently solved by an encoding to SAT; encodings are often discussed as part of the proof that a problem is a member of the NP-complete complexity class (one method of proving this is to provide a polynomial-time encoding to SAT). Despite being believed to be exponential, many instances of SAT that arise in practice turn out to be amenable to machine solving.

Definition 2.2.1 (satisfiability problem)
The *satisfiability problem* for a formula f is the problem of finding an assignment $\sigma : AP \to \{\top, \bot\}$ such that $\sigma \models f$. Such an assignment is called a *total solution* or a *total satisfying assignment*. We say that a formula f is *satisfiable* if such an interpretation exists, and *unsatisfiable* otherwise.

We also consider solutions that are partial assignments to the atomic propositions in the formula: the structure of the formula may make it unnecessary to provide assignments to all of the atomic propositions. In fact, it can be *more* useful to provide a partial assignment rather than a full one because by implicitly representing a whole set of assignments, it also indicates which atomic propositions are important in determining the truth of the formula.

Definition 2.2.2 (partial solution)
A *partial solution* for a formula f is a *partial satisfying assignment* $\sigma : AP \nrightarrow \{\top, \bot\}$ such that $\sigma \models f$.

Lemma 2.1 (partial satisfaction) *A formula f is satisfiable if a partial satisfying assignment for f exists, and unsatisfiable otherwise.*

Proof This follows directly from Definition 2.2.2. □

Where the total or partial nature of a satisfying assignment is unimportant, we will refer simply to *satisfying assignments* and *solutions*. There may be a large number of different satisfying assignments to a given problem, and the solver is under no obligation to provide any particular one; this is a problem that must be dealt with for some encodings if a notion of a *minimal* solution is required.

We will assume for most of the thesis the existence of an efficient 'black box' SAT solver: that is, we will not discuss or analyse its behaviour in mathematical terms, but simply make timing comparisons.

2.2.1 Conjunctive Normal Form

The majority of SAT solvers derived from the Davis-Putnam algorithm [34] are written to work directly on problems in *conjunctive normal form* (CNF). This is a restriction on the operators appearing in the formula and the positions in which the operators may appear: only the operators $\wedge$, $\vee$ and $\neg$ are permitted. Informally, a formula in CNF is of the form

$$(a_0 \vee a_1 \vee \cdots \vee \neg b_0 \vee \neg b_1 \vee \cdots) \wedge \cdots \wedge (c_0 \vee c_1 \vee \cdots \vee \neg d_0 \vee \neg d_1 \vee \cdots)$$

where $a_i, b_i, c_i, d_i \in AP$. That is, a conjunction of disjunctions of atomic propositions which appear either negated or unnegated.

While CNF may be seen as a formula with a particular structure, it is also convenient to view it as a set of sets of negated or unnegated atomic propositions, referred to as *clause form*. We will treat the two forms as being completely interchangeable, although the set representation removes the distinction between different orderings and eliminates duplication. From the identities in Section 2.1 we see that changes in ordering and removal of duplicates preserves the semantics of a formula. In the following text we select

the representation, either CNF or clause form, that makes the presentation most clear. Where the particular representation is not important, the terms are interchangeable. We now formally define the notion of *clause form* in terms of its constituents, and construct the corresponding CNF.

Definition 2.2.3 (literal)

A *literal* is an atomic proposition a or the negation of a atomic proposition $\neg a$ where $a \in AP$. We define the set lit of literals as

$$\mathsf{lit} \doteq AP \cup \{\neg a \mid a \in AP\}$$

Definition 2.2.4 (clause)

A *clause* is a disjunction of literals or a set of literals considered under disjunction. For example, the clause $\{a, \neg b, c\}$ represents the disjunction $a \vee \neg b \vee c$. We define the set of clauses, clause, for CNF as the smallest set obtained by the production rules

$$\frac{}{\bot \in \mathsf{clause}} \quad \frac{l \in \mathsf{lit}}{l \in \mathsf{clause}} \quad \frac{l_0 \in \mathsf{lit}, \ldots, l_n \in \mathsf{lit},\ n \geq 1}{l_0 \vee \cdots \vee l_n \in \mathsf{clause}}$$

or for clause form, clause is the set of finite sets such that

$$\mathsf{clause} \subseteq \mathsf{lit}$$

Definition 2.2.5 (CNF formula)

A formula *in CNF* is a conjunction of clauses or a set of clauses considered under conjunction. We define the set of formulae in CNF, $\mathsf{cnf} \subset \mathsf{prop}$, for CNF as the smallest set obtained by the production rules

$$\frac{}{\top \in \mathsf{cnf}} \quad \frac{c \in \mathsf{clause}}{c \in \mathsf{cnf}} \quad \frac{c_0 \in \mathsf{clause}, \ldots, c_n \in \mathsf{clause},\ n \geq 1}{c_0 \wedge \cdots \wedge c_n \in \mathsf{cnf}}$$

or for clause form, cnf is the set of finite sets such that

$$\mathsf{cnf} \subseteq \mathsf{clause}$$

We can convert between clause form and CNF by noting that a set of clauses C has the corresponding CNF formula $\bigwedge_{c \in C} \bigvee_{l \in c} l$.

Lemma 2.2 *A satisfiability problem f with $f \in \mathsf{cnf}$ is satisfied by a total assignment σ if and only if for every clause $c \in f$, there is a literal $a \in c$ such that $\langle a, \top \rangle \in \sigma$ or a literal $\neg a \in c$ such that $\langle a, \bot \rangle \in \sigma$.*

Proof This follows directly from the normal rules for propositional logic. □

Since CNF is sufficiently general to represent any formula, we may convert problems from a general propositional representation to a CNF representation. We look first at the general problem of converting propositional formulae to CNF, and the various approaches available; in Section 2.3 we look at applying these generic approaches to the more efficient representation of propositional formulae that we will be using throughout the thesis.

Definition 2.2.6 (CNF conversion)
A *CNF conversion* is a procedure to convert a general propositional formula f into an equivalent CNF formula f' such that $f \equiv f'$.

A CNF conversion is typically achieved in three stages: a preprocessing step aimed at reducing redundancy and at achieving a smaller resulting set of clauses; a reduction to negation normal form (NNF), which involves replacing Boolean connectives with their equivalents in terms of conjunction and disjunction (see Figure 2.2); and the application of identities to ensure that the structure conforms to CNF. The simplest CNF conversion is that with no preprocessing step, and we therefore begin by defining the reduction to NNF.

Definition 2.2.7 (NNF formula)
A formula f is said to be *in NNF* if the only connectives appearing in f are members of the set $\{\wedge, \vee, \neg\}$ and negations, $\neg$, apply only directly to atomic propositions. We define the set $\mathsf{nnf} \subset \mathsf{prop}$ of formulae in NNF as the smallest set obtained by the production rules

$$\frac{}{\top, \bot \in \mathsf{nnf}} \qquad \frac{l \in \mathsf{lit}}{l \in \mathsf{nnf}}$$

$$\frac{f_0 \in \mathsf{nnf}, \ldots, f_n \in \mathsf{nnf},\ n \geq 1}{f_0 \wedge \cdots \wedge f_n \in \mathsf{nnf}} \qquad \frac{f_0 \in \mathsf{nnf}, \ldots, f_n \in \mathsf{nnf},\ n \geq 1}{f_0 \vee \cdots \vee f_n \in \mathsf{nnf}}$$

Note that $\mathsf{nnf} \supset \mathsf{cnf}$.

One possible conversion to NNF is performed by the recursive function given in Figure 2.2. This function is derived from the standard definitions of Boolean connectives and the application of de Morgan's rule. The function has the effect of 'pushing' negations in towards the atomic propositions, leaving operators replaced by their equivalents in terms of $\wedge$ and $\vee$. Bi-implication ($\leftrightarrow$) is treated with a little care as there is a choice of substitutions to be made (the

alternative would be $(\text{NNF}(f) \wedge \text{NNF}(g)) \vee (\text{NNF}(\neg f) \wedge \text{NNF}(\neg g))$ in the positive case); we choose the substitution given to bring us closer to CNF during the conversion.

$$
\begin{aligned}
\text{NNF}(a) &= a \\
\text{NNF}(\neg a) &= \neg a \\
\text{NNF}(\neg\neg f) &= \text{NNF}(f) \\
\text{NNF}(f_0 \wedge f_1) &= \text{NNF}(f_0) \wedge \text{NNF}(f_1) \\
\text{NNF}(\neg(f_0 \wedge f_1)) &= \text{NNF}(\neg f_0) \vee \text{NNF}(\neg f_1) \\
\text{NNF}(f_0 \vee f_1) &= \text{NNF}(f_0) \vee \text{NNF}(f_1) \\
\text{NNF}(\neg(f_0 \vee f_1)) &= \text{NNF}(\neg f_0) \wedge \text{NNF}(\neg f_1) \\
\text{NNF}(f_0 \rightarrow f_1) &= \text{NNF}(\neg f_0) \vee \text{NNF}(f_1) \\
\text{NNF}(\neg(f_0 \rightarrow f_1)) &= \text{NNF}(f_0) \wedge \text{NNF}(\neg f_1) \\
\text{NNF}(f_0 \leftrightarrow f_1) &= (\text{NNF}(\neg f_0) \vee \text{NNF}(f_1)) \wedge (\text{NNF}(f_0) \vee \text{NNF}(\neg f_1)) \\
\text{NNF}(\neg(f_0 \leftrightarrow f_1)) &= (\text{NNF}(f_0) \vee \text{NNF}(f_1)) \wedge (\text{NNF}(\neg f_0) \vee \text{NNF}(\neg f_1))
\end{aligned}
$$

Figure 2.2: The NNF conversion function $\text{NNF}(f)$

Conversion to NNF achieves much of the CNF conversion: negations are in the right place, and the number of different types of operator has been reduced to those allowed in CNF; the only remaining step is the rearranging of the disjunctions to lie under the conjunctions. As for NNF, we push conjunctions in, closer to the literals. The functional definition given in Figure 2.3 is based on the distributive rule $f_0 \vee (f_1 \wedge f_2) \equiv (f_0 \vee f_1) \wedge (f_0 \vee f_2)$ and its equivalents. These distributive steps increase the number of connectives—in the worst case, where the original formula is in disjunctive normal form, exponentially. Consider the formula $(a_0 \wedge b_0) \vee \cdots \vee (a_n \wedge b_n)$; the resulting CNF consists of a series of n-ary disjunctions for each combination of a_i and b_i, $(a_0 \vee \cdots \vee a_n) \wedge \cdots \wedge (b_0 \vee \cdots \vee b_n)$, converting a formula with $2n - 1$ connectives to one with $2^{n+1}n - 1$ connectives.

We will refer to the CNF conversion $\text{CNF}(\text{NNF}(f))$ as the *standard* CNF conversion. The resulting problem has the same number of atomic propositions, but can have an exponentially larger number of connectives in the worst case.

$$\begin{aligned}
\textsc{Cnf}(a) &= a \\
\textsc{Cnf}(\neg a) &= \neg a \\
\textsc{Cnf}((f_0 \wedge f_1) \vee f_2) &= \textsc{Cnf}(f_0 \vee f_2) \wedge \textsc{Cnf}(f_1 \vee f_2) \\
\textsc{Cnf}(f_0 \vee (f_1 \wedge f_2)) &= \textsc{Cnf}(f_0 \vee f_1) \wedge \textsc{Cnf}(f_0 \vee f_2) \\
\textsc{Cnf}(f_0 \vee f_1) &= \textsc{Cnf}(f_0) \vee \textsc{Cnf}(f_1) \quad \text{if } f_0, f_1 \text{ are not conjunctions} \\
\textsc{Cnf}(f_0 \wedge f_1) &= \textsc{Cnf}(f_0) \wedge \textsc{Cnf}(f_1)
\end{aligned}$$

Figure 2.3: The standard CNF conversion function $\textsc{Cnf}(f)$ for $f \in$ nnf

Lemma 2.3 (equivalence of the standard CNF conversion) *For any propositional formula f, the formula obtained by the CNF conversion* $\textsc{Cnf}(\textsc{Nnf}(f))$ *is equivalent to f.*

Proof Each line in the definition of $\textsc{Cnf}(f)$ and $\textsc{Nnf}(f)$ is an equivalence following from those given in Section 2.1, so this follows by induction. The base case is the atomic proposition, for which both Cnf and Nnf are the identity function. The confluence of the transformation is assured by the distributive and associative properties given in Section 2.1. □

2.2.1.1 Improved Clause Form Conversions

The size explosion seen in the standard clause form conversion stems from the distributive rules operating over formulae. However, the result of Lemma 2.3 is stronger than is required in practice: it is sufficient that the clause form is satisfiable by the same assignments that satisfy the original formula.

Definition 2.2.8 (equisatisfiable)
A formulae f and f' are said to be *equisatisfiable*, written $f \cong f'$, if every satisfying assignment to f which does not constrain variables in $ap(f') \setminus ap(f)$ can be extended to a partial solution to f' and vice versa:

$$\forall \sigma, (dom(\sigma) \cap ap(f') \setminus ap(f) = \emptyset), \sigma \models f \;.\; \exists \sigma' \supseteq \sigma \;.\; \sigma' \models f'$$

and

$$\forall \sigma', (dom(\sigma') \cap ap(f) \setminus ap(f') = \emptyset), \sigma' \models f' \;.\; \exists \sigma \supseteq \sigma' \;.\; \sigma \models f$$

In the restricted case that $ap(f') \supseteq ap(f)$, the latter condition is equivalent to simply

$$\forall \sigma, \sigma \models f' \;.\; \sigma \models f$$

That is, satisfying assignments to f' also satisfy f. This case occurs with the equisatisfiable CNF conversions which introduce additional propositions.

We broaden the definition of CNF conversions in Definition 2.2.6 to allow for equisatisfiability.

Definition 2.2.9 (equisatisfiable CNF conversion)
An *(equisatisfiable) CNF conversion* is a procedure to convert a general propositional formula f into an equisatisfiable CNF formula f'.

The notion of equisatisfiable CNF conversions is proposed by Plaisted and Greenbaum [84]. They define a clause form conversion which involves the introduction of a new atomic proposition for each subformula, taking advantage of Tseitin's observation [98] that such substitutions can reduce the length of a proof exponentially.

The definitions of the atomic propositions as the corresponding subformulae is given as a series of conjuncts and the truth or falsity of the formula as a whole is then given by the value of the atomic proposition which represents it. This idea is called *renaming*: replacing subformulae by new atomic propositions and adding suitable definitions. For example, $(a \wedge \neg b) \vee \neg(b \wedge c)$ becomes

$$r_1 \wedge (r_1 \leftrightarrow r_2 \vee \neg r_3) \wedge (r_2 \leftrightarrow a \wedge \neg b) \wedge (r_3 \leftrightarrow b \wedge c)$$

Definition 2.2.10 (renaming)
Consider a formula $F[f] \in \mathsf{prop}$ with $r_f \notin ap(F[f])$. A *renaming* of the subformula f in $F[f]$ is given by

$$Ren(F[f], f) \cong F[r_f] \wedge (r_f \leftrightarrow f)$$

The *definitional* clause form conversion is the result of applying *Ren* at each position in the formula. This cannot be easily described with the formulation above as the position of subformulae change with each application of *Ren*. We therefore define the conversion function DEF(f) in Figure 2.2.1.1 recursively on the structure of f. We introduce a set of new variables indexed

$$\begin{aligned}
\mathrm{DEF}(f) &= r_f \wedge \mathrm{DEF}_R(f) \\
\mathrm{DEF}_R(a) &= \top \\
\mathrm{DEF}_R(\neg a) &= \top \\
\mathrm{DEF}_R(\neg\neg f) &= \mathrm{DEF}_R(f) \\
\mathrm{DEF}_R(f_0 \circ f_1) &= (r_{f_0 \circ f_1} \leftrightarrow (r_{f_0} \circ r_{f_1})) \wedge \mathrm{DEF}_R(f_0) \wedge \mathrm{DEF}_R(f_1) \\
\mathrm{DEF}_R(\neg(f_0 \circ f_1)) &= (\neg r_{\neg(f_0 \circ f_1)} \leftrightarrow \neg(r_{f_0} \circ r_{f_1})) \wedge \mathrm{DEF}_R(f_0) \wedge \mathrm{DEF}_R(f_1)
\end{aligned}$$

Figure 2.4: The definitional renaming function for CNF preprocessing $\mathrm{DEF}(f)$. r_f is an atomic proposition used to represent f, where $r_{\neg a} = \neg a$ and $r_a = a$ for $a \in AP$; the operator symbol $\circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$

by the subformula that they represent: r_{f_0} for each subformula f_0 in f. This has a significant side-effect: where a subformula is repeated, it is given the same variable, and hence the subformulae defining that variable are repeated. Although we give a formula-based definition in Figure 2.2.1.1, a purely set-based definition would, as noted in Section 2.2.1, have the result of eliminating this repetition and hence produce a smaller resulting formula.

Lemma 2.4 (equisatisfiability of the definitional CNF conversion) *For any propositional formula f, the result of the CNF conversion* $\textsc{Cnf}(\textsc{Nnf}(\mathrm{DEF}(f)))$ *is equisatisfiable to f.*

Proof First, we prove that $\mathrm{DEF}(f)$ is equisatisfiable to f. We show by induction that $\mathrm{DEF}_R(f)$ constrains r_f to be true only for those assignments that make f true. This holds trivially for the base cases and can be deduced by inspection of the truth tables for the inductive steps. Hence we conclude that $r_f \wedge \mathrm{DEF}_R(f)$ is equisatisfiable to f, and by Lemma 2.3 we can deduce the required result. □

2.2.1.2 Clause Form Conversion with Polarity

The conversion given by Plaisted and Greenbaum [84] builds on this conversion by using *polarity* [80] to differentiate between subformulae occurring *positively*

(under an even number of negations) or *negatively* (under an odd number of negations).

Definition 2.2.11 (polarity)
Consider a formula $F[f_0] \in \mathsf{prop}$. Let a be an atomic proposition not in $ap(F[f_0])$. We say that f_0 occurs with *positive polarity* if no occurrence of a in $\text{NNF}(F[a])$ is negated; we say that f_0 occurs with *negative polarity* if every occurrence of a in $\text{NNF}(F[a])$ is negated. Otherwise, we say that f_0 occurs with *zero polarity*.

Note that this definition of polarity is slightly non-standard in its use of context functions to force simultaneous consideration of all occurrences of a particular subformula. Others, for example Boy de la Tour [17], explicitly ignore multiple occurrences of a given subformula. The approach given is more suited to generalisation for Boolean circuits (see Section 2.3.1.2).

If a positive subformula f is replaced by a new atomic proposition r_f then, by Definition 2.2.11, after NNF conversion the new atomic proposition will itself appear positively. To ensure equisatisfiability, the definition of the new atomic proposition can be $r_f \to f$ (rather than the definition given above of $r_f \leftrightarrow f$). The following two lemmas formalise this.

Lemma 2.5 (monotonicity) *For any formula $F[a] \in \mathsf{prop}$, if every occurrence of a in $F[a]$ has positive polarity then $F[a]$ is monotonic in a. That is,*

$$(f_0 \to f) \to (F[f_0] \to F[f])$$

If every occurrence of a in $F[a]$ has negative polarity then $F[a]$ is anti-monotonic in a. That is,

$$(f_0 \to f) \to (F[f] \to F[f_0])$$

Proof We give a brief outline only. For the monotonic case, the only cases of interest are where the interpretation of f_0 is false and f is true (all other cases are trivial). By induction on the semantics (Definition 2.1.2) we see that, provided a only occurs positively in $F[a]$, a change in the value of a from false to true either preserves the value of $F[a]$ or changes it from false to true. Therefore $F[f_0] \to F[f]$ whenever $f_0 \to f$.

A similar argument can be made in the anti-monotonic case. □

Lemma 2.6 (positive/negative polarity renaming) *Consider a formula* $F[f] \in$ prop *with* $r_f \notin ap(F[f])$. *If every occurrence of* f *in* $F[f]$ *has positive polarity then*

$$F[f] \cong (r_f \rightarrow f) \wedge F[r_f]$$

If every occurrence of f *in* $F[f]$ *has negative polarity then*

$$F[f] \cong (f \rightarrow r_f) \wedge F[r_f]$$

Proof We outline the proof for the positive polarity case; the negative polarity case is similar.

By the definition of equisatisfiability (Defintion 2.2.8) we need to show that $(r_f \rightarrow f) \wedge F[r_f]$ is satisfiable whenever $F[f]$ is satisfiable; and that $((r_f \rightarrow f) \wedge F[r_f]) \rightarrow F[f]$ holds. For the former, we observe that we can always choose a value of r_f which is the same as the value of f; the latter follows from Lemma 2.5. □

Lemma 2.6 formalises positive and negative polarity renaming. In the zero polarity case (that is, the subformula occurs both positively and negatively) neither positive nor negative polarity renaming is available, so the standard renaming given in Definition 2.2.10 is used instead.

We follow Plaisted and Greenbaum in referring to this conversion as the *structure-preserving* CNF conversion (although the definitional conversion given above also preserves structure in the same sense). The conversion function (Figure 2.5) is given in terms of a function $\mathrm{SP}_R^p(f)$ which computes the definition of the atomic proposition r_f with polarity p. To simplify its definition we assume that r_l is synonymous with l for literal $l \in$ lit and we use proposition $r_{\neg f}$ (representing a negated formula) as a shorthand for $\neg r_f$.

Lemma 2.7 (equisatisfiability of the SP CNF conversion) *For any propositional formula* f, *the formula obtained by the CNF conversion* $\textsc{Cnf}(\textsc{Nnf}(\mathrm{SP}(f)))$ *is equisatisfiable to* f.

Proof The proof proceeds in the same way as for Lemma 2.4. □

Although the structure-preserving clause form conversion overcomes the exponential growth of the standard conversion (Figure 2.2.1), it does not generate smaller sets of clauses under all circumstances. Consider the case

$$
\begin{aligned}
\mathrm{SP}(f) &= r_f \wedge \mathrm{SP}^+_R(f) \\
\\
\mathrm{SP}^+_R(\neg f_0) &= \mathrm{SP}^-_R(f_0) \\
\mathrm{SP}^-_R(\neg f_0) &= \mathrm{SP}^+_R(f_0) \\
\mathrm{SP}^0_R(f_0) &= \mathrm{SP}^-_R(f_0) \wedge \mathrm{SP}^+_R(f_0) \\
\\
\mathrm{SP}^+_R(a) &= \top \\
\mathrm{SP}^-_R(a) &= \top \\
\mathrm{SP}^+_R(f_0 \vee f_1) &= (r_{f_0 \vee f_1} \rightarrow r_{f_0} \vee r_{f_1}) \wedge \mathrm{SP}^+_R(f_0) \wedge \mathrm{SP}^+_R(f_1) \\
\mathrm{SP}^+_R(f_0 \wedge f_1) &= (r_{f_0 \wedge f_1} \rightarrow r_{f_0} \wedge r_{f_1}) \wedge \mathrm{SP}^+_R(f_0) \wedge \mathrm{SP}^+_R(f_1) \\
\mathrm{SP}^+_R(f_0 \rightarrow f_1) &= (r_{f_0 \rightarrow f_1} \rightarrow r_{f_0} \rightarrow r_{f_1}) \wedge \mathrm{SP}^-_R(f_0) \wedge \mathrm{SP}^+_R(f_1) \\
\mathrm{SP}^+_R(f_0 \leftrightarrow f_1) &= (r_{f_0 \leftrightarrow f_1} \rightarrow (r_{f_0} \leftrightarrow r_{f_1})) \wedge \mathrm{SP}^0_R(f_0) \wedge \mathrm{SP}^0_R(f_1) \\
\mathrm{SP}^-_R(f_0 \vee f_1) &= (r_{f_0} \vee r_{f_1} \rightarrow r_{f_0 \vee f_1}) \wedge \mathrm{SP}^-_R(f_0) \wedge \mathrm{SP}^-_R(f_1) \\
\mathrm{SP}^-_R(f_0 \wedge f_1) &= (r_{f_0} \wedge r_{f_1} \rightarrow r_{f_0 \wedge f_1}) \wedge \mathrm{SP}^-_R(f_0) \wedge \mathrm{SP}^-_R(f_1) \\
\mathrm{SP}^-_R(f_0 \rightarrow f_1) &= ((r_{f_0} \rightarrow r_{f_1}) \rightarrow r_{f_0 \rightarrow f_1}) \wedge \mathrm{SP}^+_R(f_0) \wedge \mathrm{SP}^-_R(f_1) \\
\mathrm{SP}^-_R(f_0 \leftrightarrow f_1) &= ((r_{f_0} \leftrightarrow r_{f_1}) \rightarrow r_{f_0 \leftrightarrow f_1}) \wedge \mathrm{SP}^0_R(f_0) \wedge \mathrm{SP}^0_R(f_1)
\end{aligned}
$$

Figure 2.5: The structure preserving renaming function for CNF preprocessing $\mathrm{SP}(f)$. r_f is a new atomic proposition representing f, where $r_{\neg f} = \neg r_f$, $r_{\neg a} = \neg a$ and $r_a = a$ for $a \in AP$

of a formula already in conjunctive normal form. The structure-preserving conversion involves producing a new atomic proposition for each clause, with each definition taking one clause. The result is a worst-case doubling in the size of the clause form, where the standard conversion leaves the formula unchanged. For example, the two clause expression $(a \vee \neg b) \wedge \neg(b \wedge c)$ becomes the four clauses

$$r_1 \wedge (r_1 \rightarrow r_2 \vee \neg r_3) \wedge (r_2 \rightarrow a \vee \neg b) \wedge (b \wedge c \rightarrow r_3)$$

2.3 Boolean Formula Representation

Working directly with general propositional logic can be clumsy: it is difficult to apply simplifications without spending an excessive amount of time analysing the formula. As a result, a number of alternative representations have emerged which have useful properties. A common theme is the ability to share subformulae if they appear multiple times—leading to a graph-based representation—and the ability to perform logical simplifications quickly on the representation.

2.3.1 Boolean Circuits

In contrast to the formulaic representation of propositional logic normally used, Boolean circuits are much closer to an electronics view of logic. Labelled input *wires* take the place of atomic propositions and together with (possibly unlabelled) internal wires they are connected by logic *gates* which compute various logic functions. This makes it very natural for the results of sub-circuits to be shared amongst other parts of the circuit, as would be expected in the physical world.

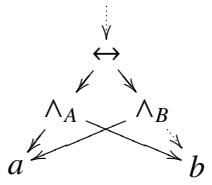
Boolean circuits may be efficiently represented as directed acyclic graphs (DAGs). Vertices having outgoing edges correspond to gates, with the edges pointing to the inputs to the gate. Vertices without outgoing edges (which we will call *leaf* vertices) are the inputs for the circuit, corresponding to atomic propositions in a propositional formula. Since edges correspond to wires the semantics of the circuit means that a formula is referred to by an edge and implicitly the graph below it. This also means that the graphs have a somewhat non-standard structure including edges with no source vertex.

Abdulla, Bjesse, and Eén proposed *reduced* Boolean circuits (RBCs) [1] as a DAG representation of a propositional formula with additional restrictions on the type and relationships of the gates which place RBCs somewhere between being a normal form and a canonical form for propositional formulae. RBCs are not canonical representations of formulae because semantically equivalent formulae may have different RBC representations if they are structured differently; however, the restrictions placed on RBCs are stronger than the syntactic restrictions usually associated with a normal form, making it a canonical representation for certain classes of formula.

One of the key strengths of Boolean circuits is the ability to use one circuit to represent a formula both positively and negatively. To preserve this property while reducing the number of possible gates, Abdulla et al. restrict gates to conjunctions and equivalences (bi-implications) and introduce the idea of marking negation on the edges of the graph. This makes it impossible to define NNF for RBCs since disjunctions are not representable without negations. Forbidding negations on the outedges of equivalences is sufficient to normalise RBCs with respect to negation: exactly one correct arrangement of negations is available for any given RBC. An ordering relation is placed on RBCs in order to allow subgraphs which differ only by the commutativity of the gates to be merged. While this ordering relation is not specified fully by Abdulla et al., implementations can use, for example, the memory addresses of vertices to achieve the necessary total order on RBCs.

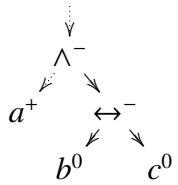
Definition 2.3.1 (RBC)
An RBC is a fragment of a DAG consisting of edges $\mathbf{E}$ and vertices $\mathbf{V} = \mathbf{V_I} \cup \mathbf{V_L}$ where internal vertices $\mathbf{V_I}$ represent operators, and leaf vertices $\mathbf{V_L}$ represent atomic propositions. The following properties are required to represent Boolean circuits as DAGs:

- Each $V \in \mathbf{V_I}$ consists of an operator $op(V) \in \{\wedge, \leftrightarrow\}$ and a left and right edge ($left(V), right(V) \in \mathbf{E}$).
- Each $V \in \mathbf{V_L}$ contains either an atomic proposition or a truth value ($var(V) \in AP \cup \{\top\}$).
- Each $E \in \mathbf{E}$ has a sign $sign(E) \in \{+, -\}$ and a target vertex $target(E) \in \mathbf{V}$.



(a) $a \wedge b \leftrightarrow (a \rightarrow b)$

represented as $\neg(a \wedge b \leftrightarrow (a \wedge \neg b))$



(b) $a \vee (b \leftrightarrow \neg c)$

represented as $\neg(\neg a \wedge (b \leftrightarrow c))$

Figure 2.6: Example RBCs showing vertex labelling

The *sign* attribute encodes negation, where $sign(E) = +$ indicates an unnegated edge and $sign(E) = -$ indicates a negated edge.

An RBC has the following additional properties which serve to reduce the number of representations possible for equivalent formulae:

- All common subformulae are shared:

$$\forall V, V' \in \mathbf{V_I}, left(V) = left(V') \wedge right(V) = right(V') \;.\; V = V'$$

- The constant $\top$ only occurs in single-vertex RBCs.
- For all vertices, $left(V) \neq right(V)$.
- If $op(V) = \leftrightarrow$ then $sign(left(V)) = sign(right(V)) = +$.
- There is a total order $\prec$ on RBCs such that for all vertices $V \in \mathbf{V_I}$ the descendants are ordered $left(V) \prec right(V)$.

We refer to an RBC formula in a DAG by its uppermost vertex or edge, since the descendant subgraph represents a complete formula. However, because of the restrictions on negation, only an edge can be used to represent general propositional formulae.

For example, Figure 2.6a shows the RBC representing the formula $a \wedge b \leftrightarrow \neg(a \rightarrow b)$. Negation is shown with a dotted edge. Some internal vertices are annotated by a subscript capital in order to allow us refer to, for example, the subformula $a \wedge b$ by the vertex A, and also to allow us to depict fragments of RBC by identifying a vertex which may have descendants without giving any further details.

2.3.1.1 RBC operations

For convenience, we will assume that one multi-rooted DAG structure is used to represent multiple RBCs with shared subgraphs. This means that forming the conjunction or equivalence of two RBCs is done in constant time by the introduction of a new vertex. Two RBCs rooted at edges L and R in the same DAG are composed by the function $rbc(L, R, o, s)$ defined below using an operation $o \in \{\wedge, \leftrightarrow\}$ and a sign $s \in \{+, -\}$:

- If o may be trivially evaluated using idempotence or the negation rule $f \wedge \neg f \equiv \bot$, return the result of doing so.
- Otherwise, check $L \prec R$ and swap if not.
- If $o = \leftrightarrow$ then s becomes $s \oplus sign(L) \oplus sign(R)$, and $sign(L)$ and $sign(R)$ become + ($\oplus$ is the exclusive-or operation).
- The new vertex V with $left(V) = L$, $right(V) = R$ and $op(V) = o$ is inserted into the DAG.
- The result is the edge E with $sign(E) = s$ and $target(E) = V$.

2.3.1.2 RBC to CNF conversion

Where a CNF conversion is given for Boolean circuits it is usual to give each wire in the circuit a corresponding atomic proposition, and encode each gate as a set of clauses constraining that atomic proposition. For example, to begin converting Figure 2.6a we would introduce atomic propositions corresponding to the vertices directly below the top level equivalence, r_A, r_B. The equivalence is represented as $\neg r_0$, and r_0 is defined in terms of r_A and r_B as $r_0 \leftrightarrow (r_A \leftrightarrow r_B)$, so we obtain the clauses $\textsc{Cnf}(\textsc{Nnf}(\neg r_0 \wedge r_0 \leftrightarrow (r_A \leftrightarrow r_B)))$.

This algorithm corresponds to the definitional CNF conversion (given in Section 2.2.1.1), and can be implemented in linear time using dynamic programming techniques. It preserves the structure of the RBC in the final CNF.

DEF(T) given in Figure 2.7 denotes the definitional clause form conversion of the subgraph beginning at an edge or vertex T. It is defined in terms of set operations for convenience—each recursive call returns the set of clauses corresponding to the subgraph starting at the function argument. Every vertex

$$\mathrm{DEF}(E) = \{\{r_E\}\} \cup \mathrm{DEF}_R(target(E))$$
$$\mathrm{DEF}(V) = \{\{r_V\}\} \cup \mathrm{DEF}_R(V)$$

$$\mathrm{DEF}_R(V) = \begin{cases} \{\} & \text{if } V \in \mathbf{V_L} \\ \textsc{Cnf}(\textsc{Nnf}(r_V \leftrightarrow (r_{left(V)} \wedge r_{right(V)}))) & \\ \quad \cup \mathrm{DEF}_R(target(left(V))) & \\ \quad \cup \mathrm{DEF}_R(target(right(V))) & \text{if } op(V) = \wedge \\ \textsc{Cnf}(\textsc{Nnf}(r_V \leftrightarrow (r_{left(V)} \leftrightarrow r_{right(V)}))) & \\ \quad \cup \mathrm{DEF}_R(target(left(V))) & \\ \quad \cup \mathrm{DEF}_R(target(right(V))) & \text{if } op(V) = \leftrightarrow \end{cases}$$

Figure 2.7: The definitional clause form conversion for RBCs. r_V is a new proposition representing vertex V, where $r_V = var(V)$ for $V \in \mathbf{V_L}$ and r_E is $r_{target(E)}$ for positive E and $\neg r_{target(V)}$ for negative E

$V \in \mathbf{V}$ is given a representative symbol r_V with the definitional clauses produced by $\textsc{Cnf}(\textsc{Nnf}(r_V \leftrightarrow a \wedge b))$ or $\textsc{Cnf}(\textsc{Nnf}(r_V \leftrightarrow (a \leftrightarrow b)))$ (see Figures 2.2 and 2.2.1) depending on the operator, where

$$\textsc{Cnf}(\textsc{Nnf}(r_V \leftrightarrow a \wedge b)) \equiv \{\{\neg r_V, a\}, \{\neg r_V, b\}, \{r_V, \neg a, b\}\}$$
$$\textsc{Cnf}(\textsc{Nnf}(r_V \leftrightarrow (a \leftrightarrow b))) \equiv \{\{\neg r_V, a, \neg b\}, \{\neg r_V, \neg a, b\}, \{r_V, a, b\}, \{r_V, \neg a, \neg b\}\}$$

We write r_E as a shorthand for the variable representing the target of E with the sign adjusted according to the $sign(E)$. This is the clause form conversion used in NuSMV [23] and BCZChaff [65].

This conversion produces a set of clauses which closely reflect the structure of the RBC: each node becomes a distinct set of three or four clauses. The CNF differs in two important ways from the strict application of the definitional conversion to the original formula. Firstly, since the RBC representation identifies repeated subformulae, the size of the resulting clause form will be reduced. Secondly, because RBCs can only represent binary connectives (gates with exactly two inputs), n-way connectives are broken down into $n - 1$ binary connectives, so introducing $n - 1$ new atomic propositions and their definitions

where the original formulation would have required only one.

This latter point is important when considering the worst case size of the clause form. Recall from above that applying the definitional conversion to a formula already in conjunctive normal form will result in a doubling of the number of clauses. For the RBC, three clauses will be generated for each binary connective needed to represent the original formula. A CNF formula with n clauses of size m will therefore grow to $3m(n-1)$ clauses. Nevertheless, this is the clause form conversion used in NuSMV, and it can also be seen in the more recent Boolean circuit work of Junttila and Niemelä [66].

We will look at the potential for applying the more advanced conversion procedures in Chapter 6.

2.4 State-Transition Systems

In this thesis we are concerned with the models of systems with a particular behaviour over time which interact with their environments. In general, such systems are captured by considering the set of possible *states* that the system can be in. The system's behaviour as a result of some action is modelled as a pair of states: the state before the action occurred and the state after it occurred; this is a transition. Computations carried out by the system are sequences of states connected by transitions.

These systems are called state-transition systems and we identify below the two particular types of state-transition system that will be considered here.

2.4.1 Büchi Automata

Büchi automata are state-transition systems with labelled transitions. The sequence of labels seen during a run of a Büchi automaton is called the *input sequence*, and to accept an input sequence, the automaton must visit a defined subset of states called the *accepting* states infinitely often. Büchi automata can therefore only recognise infinite sequences. A Büchi automata can be designed to recognise exactly the paths that satisfy any given LTL formula (see Section 2.5); we discuss LTL to Büchi automaton translations and their uses in Chapter 8.

Definition 2.4.1 (Büchi automaton)
A Büchi automaton $\mathcal{B}$ is defined by the tuple $\langle Q, \Sigma, \delta, I, T \rangle$ where Q is the set of states; Σ is the alphabet of transition labels; δ is the transition function $Q \to 2^{2^{\Sigma} \times Q}$; $I \subseteq Q$ is the set of initial states; $T \subseteq Q$ is the set of accepting states.

A run of a Büchi automaton is a path through the automaton; it is accepting if the states in T are visited an infinite number of times.

Definition 2.4.2 (accepting run)
A *run* of a Büchi automaton $\mathcal{B}$ with respect to a word $u_0 u_1 \ldots \in \Sigma^{\omega}$ is a sequence of states $q_0 q_1 \ldots \in Q^{\omega}$ with $q_0 \in I$ and $\forall i \exists \alpha_i \,.\, \langle \alpha_i, q_{i+1} \rangle \in \delta(q_i)$ such that $u_i \in \alpha_i$. A run is *accepting* if infinitely many states in the run are members of T.

A *generalised* Büchi automaton (GBA) has a finite set of accepting sets $\mathcal{T} \subseteq 2^{Q}$; each set must be visited infinitely often for acceptance. A GBA may be reduced to a classical Büchi automaton but incurs a linear blowup of $O(|\mathcal{T}|)$.

2.4.2 Kripke Structures

Kripke structures are state transition systems with a labelling function identifying the set of atomic propositions which are true in each state. They are defined in detail below. However, in practice it is unusual to submit a Kripke structure directly to a model checker. Instead, a high-level language is used to define a *symbolic Kripke structure* with a transition relation given in terms of *state variables*, whose values uniquely identify each state, and over which the set of propositions used in the labelling function are defined. State variables may be Boolean variables, integer variables, or higher-level constructs such as arrays, but for the purpose of this thesis we will assume that all variables have been encoded to Booleans. The way that this is done is out of scope for this work; however, see Section 10.1.

Definition 2.4.3 (symbolic Kripke structure)
A *symbolic Kripke structure*, $\hat{M}$ is a tuple $\langle A, \hat{I}(A), \hat{T}(A, A') \rangle$ where A is a set of atomic propositions. We write A' for a copy of A of the form $\{a' \mid a \in A\}$ and A^i with $i \in \mathbb{N}$ for a copy $\{a^i \mid a \in A\}$.

$\hat{I}(A)$ and $\hat{T}(A, A')$ are symbolic representations of formulae in propositional logic which are equivalent to true when, for $\hat{I}(A)$, the truth-value assignment for A describes an initial state, or for $\hat{T}(A, A')$, the truth-value assignment for $A \cup A'$ describes a pair of states connected by a transition.

We write $\hat{I}(A^i)$ for $\hat{I}(A)$ with each $a \in A$ replaced by $a^i \in A^i$, and $\hat{T}(A^i, A^j)$ for $\hat{T}(A, A')$ with each $a \in A^i$ replaced by $a^i \in A$, and each $a' \in A'$ replaced by $a^j \in A^j$.

The symbolic representations of $\hat{I}(A)$ and $\hat{T}(A, A')$ could be in any appropriate form, such as a BDD or RBC. While we will consider symbolic Kripke structures to be the primary representation in this thesis, some of the theoretical work requires the corresponding semantic Kripke structure.

Definition 2.4.4 (Kripke structure)
A *Kripke structure* is a tuple $\langle S, T, L, I \rangle$ where S is a set of states; $T \subseteq S \times S$ is the transition relation, which is required to be total ($dom(T) = S$); $L : S \to 2^{AP}$ is the labelling function, marking each state with the set of atomic propositions (AP) that hold in that state; and I is the set of initial states.

A sequence of states connected by transitions through a Kripke structure is called a *path*. We restrict the paths considered to be those which begin the set of initial states I.

Definition 2.4.5 (path)
A *path π through a Kripke structure M*, written $\pi \in M$ is a path $s_0, s_1, \ldots \in S$ such that $s_0 \in I$, and for all $i \in \mathbb{N}$, $\langle s_i, s_{i+1} \rangle \in T$. We write $\pi(i) = s_i$ to refer to individual states within a path.

A symbolic Kripke structure $\hat{M}$ has the corresponding semantic Kripke structure $K(\hat{M})$ given by

$$K(\hat{M}) = \langle 2^A, \{\langle s, s' \rangle \mid s, s' \models \hat{T}(A, A')\}, \{\langle s, s \rangle \mid s \in 2^A\}, \{\langle s \rangle \mid s \models \hat{I}(A)\} \rangle$$

2.5 Temporal Logic

Temporal logic extends conventional logic to include reasoning about time. Whilst propositional logic is concerned with a fixed interpretation of a set of

propositional symbols, temporal logic reinterprets the symbols with respect to time, modelled as a sequence of discrete states.

Clearly, the structure of time is significant in the understanding of temporal logic. For this thesis we focus on *linear* temporal logic: logic in which time is considered to be a linear sequence of states, each having only one future and one past. An alternative, *branching time* temporal logic, considers the possibility of a state having multiple futures; a tree structure which may more adequately describe properties of some systems. We discuss the branching time logic CTL in Section 2.5.2 briefly as it is the primary specification language used in symbolic model checking (see Section 2.6.1). There is an extensive literature on the whether linear time or branching time logic is preferred in model checking; see for example Vardi [99].

We define LTL with respect to a symbolic Kripke structure, which determines the set of atomic propositions AP and the definition of a path.

The language of the linear temporal logic *LTL* [85] is a superset of propositional logic.

Definition 2.5.1 (LTL formula)

The set $\mathsf{ltl} \supset \mathsf{prop}$ of formulae in LTL is the smallest set obtained by the production rules

$$\frac{}{\top, \bot \in \mathsf{ltl}} \qquad \frac{a \in AP}{a \in \mathsf{ltl}} \qquad \frac{\phi \in \mathsf{ltl}}{\neg\phi \in \mathsf{ltl}}$$

$$\frac{\phi_0, \ldots, \phi_n \in \mathsf{ltl},\ n \geq 1}{\phi_0 \wedge \cdots \wedge \phi_n \in \mathsf{ltl}} \qquad \frac{\phi_0, \ldots, \phi_n \in \mathsf{ltl},\ n \geq 1}{\phi_0 \vee \cdots \vee \phi_n \in \mathsf{ltl}} \qquad \frac{\phi_0, \phi_1 \in \mathsf{ltl}}{\phi_0 \rightarrow \phi_1 \in \mathsf{ltl}} \qquad \frac{\phi_0, \phi_1 \in \mathsf{ltl}}{\phi_0 \leftrightarrow \phi_1 \in \mathsf{ltl}}$$

$$\frac{\phi \in \mathsf{ltl}}{\mathbf{X}\,\phi \in \mathsf{ltl}} \qquad \frac{\phi \in \mathsf{ltl}}{\mathbf{F}\,\phi \in \mathsf{ltl}} \qquad \frac{\phi \in \mathsf{ltl}}{\mathbf{G}\,\phi \in \mathsf{ltl}} \qquad \frac{\phi_0, \phi_1 \in \mathsf{ltl}}{\phi_0\,\mathbf{U}\,\phi_1 \in \mathsf{ltl}} \qquad \frac{\phi_0, \phi_1 \in \mathsf{ltl}}{\phi_0\,\mathbf{R}\,\phi_1 \in \mathsf{ltl}}$$

It is usual to give semantics of temporal logics using general first-order logic. To help clarify the translations given in the following chapter, we make explicit the fragment of first-order logic that will be required. In the logic *QTPL* (propositional logic with quantification over time), all variables are natural numbers so quantifiers are only written in terms of linear constraints on these variables. The set of predicates is restricted to monadic predicates parameterised by natural numbers. In the treatment of the semantics of LTL given below, time steps along a path are mapped to the integers, and atomic propositions in LTL are mapped to monadic predicates where the parameter of the predicate indicates the time step along the path.

Definition 2.5.2 (QTPL formula)

Propositional logic with quantification over time (QTPL) is a restriction of first-order logic. Let N be the a set of variables ranging over integers, and LC be the language of linear constraints over the variables in N, defined as the smallest set obtained from

$$\frac{i, j \in N}{i < j \in LC} \quad \frac{i, j \in N}{i = j \in LC} \quad \frac{f \in LC}{\neg f \in LC}$$

$$\frac{f_0, \ldots, f_n \in LC,\ n \geq 1}{f_0 \wedge \cdots \wedge f_n \in LC} \quad \frac{f_0, \ldots, f_n \in LC,\ n \geq 1}{f_0 \vee \cdots \vee f_n \in LC}$$

Then the set of formulae in $\mathsf{qtpl} \supset \mathsf{prop}$ is the smallest set obtained from the productions below.

$$\frac{}{\top, \bot \in \mathsf{qtpl}} \quad \frac{a \in AP, i \in N}{a(i) \in \mathsf{qtpl}} \quad \frac{f \in \mathsf{qtpl}}{\neg f \in \mathsf{qtpl}}$$

$$\frac{f_0, \ldots, f_n \in \mathsf{qtpl},\ n \geq 1}{f_0 \wedge \cdots \wedge f_n \in \mathsf{qtpl}} \quad \frac{f_0, \ldots, f_n \in \mathsf{qtpl},\ n \geq 1}{f_0 \vee \cdots \vee f_n \in \mathsf{qtpl}}$$

$$\frac{f_0, f_1 \in \mathsf{qtpl}}{f_0 \rightarrow f_1 \in \mathsf{qtpl}} \quad \frac{f_0, f_1 \in \mathsf{qtpl}}{f_0 \leftrightarrow f_1 \in \mathsf{qtpl}}$$

$$\frac{f \in \mathsf{qtpl},\ i \in N,\ c \in LC}{\exists i, c\ .\ f \in \mathsf{qtpl}} \quad \frac{f \in \mathsf{qtpl},\ i \in N,\ c \in LC}{\forall i, c\ .\ f \in \mathsf{qtpl}}$$

We will also allow the usual syntactic sugaring of quantifiers and their arguments, for example writing $\forall x, y \ldots$ for $\forall x\ .\ \forall y \ldots$, and of the linear constraints to allow the use of $\leq$, $\geq$, etc.

Unlike some formulations of LTL (for example, see Clarke et al. [27]), we do not refer to path suffixes but instead include path offsets in the definition of the semantic interpretation operator. LTL is defined over infinite paths, although not every LTL operator is infinite in its scope.

Definition 2.5.3 (LTL semantics)

An infinite path π in a Kripke structure M satisfies an LTL formula ϕ at a position i, written $\pi \models^i \phi$, if the sequence of states $\pi(i), \pi(i+1), \ldots$ is consistent with ϕ, as given by Figure 2.8.

We abbreviate $\models^0$ as $\models$.

For clarity, we also give an informal semantics for the additional operators seen in LTL:

- $\mathbf{X}\,\phi$ ("next") holds if ϕ holds in the next moment

$$
\begin{array}{lcl}
\pi \models^i a & \Leftrightarrow & a \in \pi(i) \\
\pi \models^i \neg\phi & \Leftrightarrow & \pi \not\models^i \phi \\
\pi \models^i \phi \wedge \psi & \Leftrightarrow & (\pi \models^i \phi) \wedge (\pi \models^i \psi) \\
\pi \models^i \phi \vee \psi & \Leftrightarrow & (\pi \models^i \phi) \vee (\pi \models^i \psi) \\
\pi \models^i \mathbf{X}\,\phi & \Leftrightarrow & \pi \models^{i+1} \phi \\
\pi \models^i \mathbf{F}\,\phi & \Leftrightarrow & \exists j, i \leq j \;.\; \pi \models^j \phi \\
\pi \models^i \mathbf{G}\,\phi & \Leftrightarrow & \forall j, i \leq j \;.\; \pi \models^j \phi \\
\pi \models^i \phi\,\mathbf{U}\,\psi & \Leftrightarrow & \exists j, i \leq j \;.\; \pi \models^j \psi \wedge \forall n, i \leq n < j \;.\; \pi \models^n \phi \\
\pi \models^i \phi\,\mathbf{R}\,\psi & \Leftrightarrow & \forall j, i \leq j \;.\; \pi \models^j \psi \vee \exists n, i \leq n < j \;.\; \pi \models^n \phi
\end{array}
$$

Figure 2.8: The semantics of LTL

- $\mathbf{F}\,\phi$ ("finally") holds if ϕ holds at some time in the future
- $\mathbf{G}\,\phi$ ("globally") holds if ϕ holds in all future moments
- $\phi\,\mathbf{U}\,\psi$ ("until") holds if ψ holds at some time in the future and ϕ holds in all moments from now until that time
- $\phi\,\mathbf{R}\,\psi$ ("release") holds if, for every continuous sequence of states starting from now in which ϕ does not hold, ψ holds in the following moment

X is known as a *step* operator as it refers only to states a single step away; the other operators are called *infinite time* operators. We will refer to the **F** operator as an *eventuality*. We define the *weak until* operator, $\phi\,\mathbf{W}\,\psi$, as $\phi\,\mathbf{W}\,\psi \doteq (\phi\,\mathbf{U}\,\psi) \vee \mathbf{G}\,\phi$, and notice that $\phi\,\mathbf{U}\,\psi \equiv \phi\,\mathbf{W}\,\psi \wedge \mathbf{F}\,\psi$. The following logical duals are identified:

$$
\begin{array}{ll}
\neg\,\mathbf{X}\,\phi \equiv \mathbf{X}\,\neg\phi & \neg(\phi\,\mathbf{U}\,\psi) \equiv \neg\phi\,\mathbf{R}\,\neg\psi \\
\neg\,\mathbf{F}\,\phi \equiv \mathbf{G}\,\neg\phi & \neg(\phi\,\mathbf{R}\,\psi) \equiv \neg\phi\,\mathbf{U}\,\neg\psi \\
\neg\,\mathbf{G}\,\phi \equiv \mathbf{F}\,\neg\phi &
\end{array}
$$

We extend several concepts defined above for propositional logic to apply to LTL. Negation normal form (see Definition 2.2.7) is applied to LTL in Figure 2.9 to form $\mathsf{nnf}_l \subset \mathsf{ltl}$. Substitutions and context functions (see Sec-

tion 2.1.1) are defined for LTL in Section 2.5.3. The definition of polarity (Definition 2.2.11) then extends trivially to LTL.

$$
\begin{aligned}
\textsc{Nnf}(a) &= a \\
\textsc{Nnf}(\neg a) &= \neg a \\
\textsc{Nnf}(\varphi \wedge \psi) &= \textsc{Nnf}(\varphi) \wedge \textsc{Nnf}(\psi) \\
\textsc{Nnf}(\neg(\varphi \wedge \psi)) &= \textsc{Nnf}(\neg\varphi) \vee \textsc{Nnf}(\neg\psi) \\
\textsc{Nnf}(\varphi \vee \psi) &= \textsc{Nnf}(\varphi) \vee \textsc{Nnf}(\psi) \\
\textsc{Nnf}(\neg(\varphi \vee \psi)) &= \textsc{Nnf}(\neg\varphi) \wedge \textsc{Nnf}(\neg\psi) \\
\textsc{Nnf}(\varphi \rightarrow \psi) &= \textsc{Nnf}(\neg\varphi) \vee \textsc{Nnf}(\psi) \\
\textsc{Nnf}(\neg(\varphi \rightarrow \psi)) &= \textsc{Nnf}(\varphi) \wedge \textsc{Nnf}(\neg\psi) \\
\textsc{Nnf}(\varphi \leftrightarrow \psi) &= (\textsc{Nnf}(\neg\varphi) \vee \textsc{Nnf}(\psi)) \wedge (\textsc{Nnf}(\varphi) \vee \textsc{Nnf}(\neg\psi)) \\
\textsc{Nnf}(\neg(\varphi \leftrightarrow \psi)) &= (\textsc{Nnf}(\varphi) \vee \textsc{Nnf}(\psi)) \wedge (\textsc{Nnf}(\neg\varphi) \vee \textsc{Nnf}(\neg\psi)) \\
\textsc{Nnf}(\mathbf{X}\,\phi) &= \mathbf{X}\,\textsc{Nnf}(\phi) \\
\textsc{Nnf}(\neg\,\mathbf{X}\,\phi) &= \mathbf{X}\,\textsc{Nnf}(\neg\phi) \\
\textsc{Nnf}(\mathbf{F}\,\phi) &= \mathbf{F}\,\textsc{Nnf}(\phi) \\
\textsc{Nnf}(\neg\,\mathbf{F}\,\phi) &= \mathbf{G}\,\textsc{Nnf}(\neg\phi) \\
\textsc{Nnf}(\mathbf{G}\,\phi) &= \mathbf{G}\,\textsc{Nnf}(\phi) \\
\textsc{Nnf}(\neg\,\mathbf{G}\,\phi) &= \mathbf{F}\,\textsc{Nnf}(\neg\phi) \\
\textsc{Nnf}(\phi\,\mathbf{U}\,\psi) &= \textsc{Nnf}(\phi)\,\mathbf{U}\,\textsc{Nnf}(\psi) \\
\textsc{Nnf}(\neg(\phi\,\mathbf{U}\,\psi)) &= \textsc{Nnf}(\neg\phi)\,\mathbf{R}\,\textsc{Nnf}(\psi) \\
\textsc{Nnf}(\phi\,\mathbf{R}\,\psi) &= \textsc{Nnf}(\phi)\,\mathbf{R}\,\textsc{Nnf}(\psi) \\
\textsc{Nnf}(\neg(\phi\,\mathbf{R}\,\psi)) &= \textsc{Nnf}(\neg\phi)\,\mathbf{U}\,\textsc{Nnf}(\psi)
\end{aligned}
$$

Figure 2.9: The NNF conversion function $\textsc{Nnf}(\phi)$ extended to LTL

2.5.1 LTL with Past Operators

In the semantics given above, every temporal operator, evaluated at time i, is defined in terms of times greater than or equal to i. We call such operators *future time* operators. For every future time operator, we can define a *past time*

operator with similar semantics, but looking to times less than or equal to i. An important difference with past time operators is the 'boundary' effect of the initial state: in the future time semantics, we can assume that every state has a successor; in the past time semantics, every state except for the initial state has a predecessor. For this reason, we have two past time step operators—logical duals of one another— which evaluate to true or false in the initial state.

Definition 2.5.4 (PLTL formula)

The set $\mathsf{pltl} \supset \mathsf{ltl}$ of formulae in PLTL is the smallest set obtained by the production rules

$$\frac{}{\top, \bot \in \mathsf{pltl}} \quad \frac{a \in AP}{a \in \mathsf{pltl}} \quad \frac{\phi \in \mathsf{pltl}}{\neg\phi \in \mathsf{pltl}}$$

$$\frac{\phi_0, \ldots, \phi_n \in \mathsf{pltl},\ n \geq 1}{\phi_0 \wedge \cdots \wedge \phi_n \in \mathsf{pltl}} \quad \frac{\phi_0, \ldots, \phi_n \in \mathsf{pltl},\ n \geq 1}{\phi_0 \vee \cdots \vee \phi_n \in \mathsf{pltl}}$$

$$\frac{\phi_0, \phi_1 \in \mathsf{pltl}}{\phi_0 \rightarrow \phi_1 \in \mathsf{pltl}} \quad \frac{\phi_0, \phi_1 \in \mathsf{pltl}}{\phi_0 \leftrightarrow \phi_1 \in \mathsf{pltl}}$$

$$\frac{\phi \in \mathsf{pltl}}{\mathbf{X}\phi \in \mathsf{pltl}} \quad \frac{\phi \in \mathsf{pltl}}{\mathbf{F}\phi \in \mathsf{pltl}} \quad \frac{\phi \in \mathsf{pltl}}{\mathbf{G}\phi \in \mathsf{pltl}} \quad \frac{\phi_0, \phi_1 \in \mathsf{pltl}}{\phi_0 \,\mathbf{U}\, \phi_1 \in \mathsf{pltl}} \quad \frac{\phi_0, \phi_1 \in \mathsf{pltl}}{\phi_0 \,\mathbf{R}\, \phi_1 \in \mathsf{pltl}}$$

$$\frac{\phi \in \mathsf{pltl}}{\mathbf{Y}\phi \in \mathsf{pltl}} \quad \frac{\phi \in \mathsf{pltl}}{\mathbf{Z}\phi \in \mathsf{pltl}} \quad \frac{\phi \in \mathsf{pltl}}{\mathbf{O}\phi \in \mathsf{pltl}} \quad \frac{\phi \in \mathsf{pltl}}{\mathbf{H}\phi \in \mathsf{pltl}}$$

$$\frac{\phi_0, \phi_1 \in \mathsf{pltl}}{\phi_0 \,\mathbf{S}\, \phi_1 \in \mathsf{pltl}} \quad \frac{\phi_0, \phi_1 \in \mathsf{pltl}}{\phi_0 \,\mathbf{T}\, \phi_1 \in \mathsf{pltl}}$$

As before, we give an informal semantics for the additional operators seen in PLTL:

- $\mathbf{Y}\,\phi$ ("yesterday") holds if there is a previous moment and ϕ holds in it
- $\mathbf{Z}\,\phi$ ("previous") holds if ϕ holds in the previous moment, and also at the start of time
- $\mathbf{O}\,\phi$ ("once") holds if ϕ holds at some time in the past
- $\mathbf{H}\,\phi$ ("historically") holds if ϕ holds in all earlier moments
- $\phi\,\mathbf{S}\,\psi$ ("since") holds if ϕ holds now and in all earlier moments since ψ holds
- $\phi\,\mathbf{T}\,\psi$ ("trigger") holds if ψ holds now and in all earlier moments until (and including) the next occurrence of ϕ, or in all earlier moments

$$
\begin{aligned}
\pi \models^i \mathbf{Y}\,\phi &\quad\Leftrightarrow\quad i > 0 \wedge \pi \models^{i-1} \phi \\
\pi \models^i \mathbf{Z}\,\phi &\quad\Leftrightarrow\quad i = 0 \vee \pi \models^{i-1} \phi \\
\pi \models^i \mathbf{O}\,\phi &\quad\Leftrightarrow\quad \exists j \leq i \,.\, \pi \models^j \phi \\
\pi \models^i \mathbf{H}\,\phi &\quad\Leftrightarrow\quad \forall j \leq i \,.\, \pi \models^j \phi \\
\pi \models^i \phi\,\mathbf{S}\,\psi &\quad\Leftrightarrow\quad \exists j \leq i \,.\, (\pi \models^j \psi \wedge \forall k, j < k \leq i \,.\, \pi \models^k \phi) \\
\pi \models^i \phi\,\mathbf{T}\,\psi &\quad\Leftrightarrow\quad \forall j \leq i \,.\, (\pi \models^j \psi \vee \exists k, j < k \leq i \,.\, \pi \models^k \phi)
\end{aligned}
$$

Figure 2.10: The semantics of the PLTL past time operators

2.5.2 Computational Tree Logic

We define the branching time logic only informally: it is relevant to the comparison between traditional model checking and bounded model checking, but will not form part of the mathematical discourse. CTL extends LTL by introducing the universal and existential quantifiers over possible future paths **A** and **E** which appear directly before a temporal operator.

- $\mathbf{A}\,\phi$ holds if ϕ holds in all possible future paths
- $\mathbf{E}\,\phi$ holds if there exists a future path in which ϕ holds

For CTL, each ϕ above must have an LTL operator is its main connective; the generalisation CTL*[44] allows for any ϕ. While there are LTL properties which are inexpressible in CTL and vice versa, CTL* is a superset of both languages. For example, $\mathbf{AF}\,\mathbf{EX}\,\bot$ is a CTL (and CTL*) expression for the reachability of a deadlock state, not expressible in LTL.

2.5.3 Context Functions in LTL

We extend the idea of context functions 2.1.1 to cover LTL, firstly extending substitution (Definition 2.1.5) to cover LTL operators.

Definition 2.5.5 (substitution)
A substitution $\phi[\psi/a]$ is the replacement of every occurrence of a in ϕ by ψ. Formally,

$$
\begin{aligned}
a[\psi/a] &\equiv \psi \\
v[\psi/a] &\equiv v \qquad\qquad v \in AP, v \neq \psi
\end{aligned}
$$

$$
\begin{aligned}
(\neg\phi_0)[\psi/a] &\equiv \neg(\phi_0[\psi/a]) \\
(\phi_0 \wedge \phi_1)[\psi/a] &\equiv (\phi_0[\psi/a]) \wedge (\phi_1[\psi/a]) \\
(\phi_0 \vee \phi_1)[\psi/a] &\equiv (\phi_0[\psi/a]) \vee (\phi_1[\psi/a]) \\
(\phi_0 \rightarrow \phi_1)[\psi/a] &\equiv (\phi_0[\psi/a]) \rightarrow (\phi_1[\psi/a]) \\
(\phi_0 \leftrightarrow \phi_1)[\psi/a] &\equiv (\phi_0[\psi/a]) \leftrightarrow (\phi_1[\psi/a]) \\
(\mathbf{X}\,\phi_0)[\psi/a] &\equiv \mathbf{X}(\phi_0[\psi/a]) \\
(\mathbf{F}\,\phi_0)[\psi/a] &\equiv \mathbf{F}(\phi_0[\psi/a]) \\
(\mathbf{G}\,\phi_0)[\psi/a] &\equiv \mathbf{G}(\phi_0[\psi/a]) \\
(\phi_0\,\mathbf{U}\,\phi_1)[\psi/a] &\equiv (\phi_0[\psi/a])\,\mathbf{U}(\phi_1[\psi/a]) \\
(\phi_0\,\mathbf{R}\,\phi_1)[\psi/a] &\equiv (\phi_0[\psi/a])\,\mathbf{R}(\phi_1[\psi/a])
\end{aligned}
$$

Definition 2.5.6 (context function)
A *context function* $\Psi[\_]$ is a formula in LTL (Definition 2.5.1) extended with the atomic proposition $\_$ (*holes*). An instantiation $\Psi[\phi]$ denotes the replacement of every hole by the formula ϕ and may be seen as a short-hand for $(\Psi[\_])[\phi/\_]$.

As before, we use syntactic pattern matching to define context functions, and make the assumption that the definition is maximal (always defines the context function with the highest number of holes).

2.5.4 Fixpoint Formulations of LTL

All LTL operators can be represented as the fixpoint of a recursive function [43]. Identifying each LTL formula ϕ with the set of paths in which is holds—the set $\{\pi \mid \pi \models \phi\}$—and defining a partial order $\langle \mathsf{ltl}, \sqsubseteq \rangle$ such that $(\phi_0 \sqsubseteq \phi_1) \Leftrightarrow \{\pi \mid \pi \models \phi_0\} \subseteq \{\pi \mid \pi \models \phi_1\}$ we can define the fixpoint operators as follows.

Definition 2.5.7 (fixpoint)
A *fixpoint* of the LTL context function $\Psi[\_]$ is an LTL formula ϕ such that $\Psi[\phi] \equiv \phi$.

Definition 2.5.8 (greatest fixpoint)
The *greatest* fixpoint $\nu Z\ .\ \Psi[Z]$ is an LTL formula $\phi = \Psi[\phi]$ such that any other fixpoint $\phi' = \Psi[\phi']$ obeys the property $\phi' \sqsubseteq \phi$.

Definition 2.5.9 (least fixpoint)
The *least* fixpoint $\nu Z \,.\, \Psi[Z]$ is an LTL formula $\phi = \Psi[\phi]$ such that any other fixpoint $\phi' = \Psi[\phi']$ obeys the property $\phi \sqsubseteq \phi'$.

The following fixpoint identities are easy to prove (see Clarke et al. [27] for proofs of the equivalent identities for CTL operators).

$$\mathbf{F}\,\phi = \mu Z \,.\, \phi \vee \mathbf{X}\,Z$$
$$\mathbf{G}\,\phi = \nu Z \,.\, \phi \wedge \mathbf{X}\,Z$$
$$\phi\,\mathbf{U}\,\psi = \mu Z \,.\, \psi \vee (\phi \wedge \mathbf{X}\,Z)$$
$$\phi\,\mathbf{R}\,\psi = \nu Z \,.\, \psi \wedge (\phi \vee \mathbf{X}\,Z)$$

2.6 Model Checking

Model checking is the process of determining whether the behaviour of a model of an evolving system, such as a transition system, fulfils a given property, such as a formula in temporal logic. The result of executing a model checker on such a problem is an indication that the model is correct or that a violation of the specification had been found; in the case of LTL model checking, an illustration of incorrectness, such as a non-compliant run of the model, can be produced.

In this thesis we are mostly concerned with LTL model checking, where the model is given as a Kripke structure and the property as an LTL formula defined over the same atomic propositions.

Definition 2.6.1 (universal LTL model checking)
The *universal LTL model checking problem* for a model M and an LTL formula ϕ defined over the same atomic propositions is to determine whether all paths $\pi \in M$ satisfy ϕ. That is, to decide

$$\forall \pi \in M \,.\, \pi \models \phi$$

Definition 2.6.2 (existential LTL model checking)
The *existential LTL model checking problem* for a model M and an LTL formula ϕ defined over the same atomic propositions is to determine whether any path $\pi \in M$ satisfies ϕ. That is, to decide

$$\exists \pi \in M \,.\, \pi \models \phi$$

The universal and existential model checking problems are logical duals: the universal problem $\forall \pi \in M \;.\; M \models \phi$ is equivalent to the existential problem $\exists \pi \in M \;.\; M \models \neg\phi$. It is usual for a universal LTL model checking procedure to return a counterexample when the model does not satisfy the formula, and for an existential procedure to similarly return a witness when the model does satisfy the formula. The counterexample to a universal problem is the same as the witness to the dual existential problem.

In subsequent chapters we will consider only the existential model checking problem.

2.6.1 Symbolic Model Checking

Symbolic model checking [21, 77] was the first method able to check models with over 10^{20} states. It works by representing sets of states in the model as BDDs. This representation is then restricted to the set of states in which each subformula of the negated specification holds by using the fixpoint representations of the temporal operators. The canonical nature of BDDs means that if the result is nonempty, it represents the states in which the specification does not hold.

This state-based, rather than path-based, nature of symbolic model checking makes it naturally applicable to CTL, rather than LTL, model checking. Methods for rewriting the LTL formula to make symbolic model checking applicable do exist, and will be discussed in Chapter 8.

2.6.2 Fairness

Some important properties of models are inexpressible in CTL, and hence unavailable to symbolic model checking without explicit extensions to the algorithm. In particular, Büchi conditions, that a state appears infinitely often on every valid path through a model, are in CTL*. As they can express properties such as the fair operation of arbiters or protocols, this particular subset of CTL* turns out to be particularly useful and is incorporated into symbolic model checkers as *fairness*. A fairness constraint on a Kripke structure is a set of states with the assertion that every path considered visits a member of the set infinitely often.

Definition 2.6.3 (fair Kripke structure)
A *fair Kripke structure* is a tuple $\langle S, T, L, I, \mathcal{F} \rangle$ where S, T, L, I are as described in Definition 2.4.4 and $\mathcal{F} \subseteq 2^S$ is the set of fairness constraints. A path $s_0, s_1, \ldots \in S$ through a fair Kripke structure is a path through the corresponding Kripke structure $\langle S, T, L, I \rangle$ (Definition 2.4.5) such that for each $F \in \mathcal{F}$, each state s_i for $i \in \mathbb{N}$ is followed eventually by a state $s_j \in F$, $j > i$.

Symbolic fair Kripke structures are defined in the same way as for standard Kripke structures in Definition 2.4.3; we do not defined a symbolic form for $\mathcal{F}$. A path through a fair Kripke structure visits a state from each fairness constraint infinitely often along the path.

We write fair model checking, the model checking problem restricted to these *fair paths*, as $M \models_F \phi$.

2.7 Summary

Most of the definitions in this chapter are standard, but a few include restrictions and notation that has an impact later in the thesis.

Symbolic Kripke structures (Section 2.4.2). Kripke structures are standard notation for model checking. We extend the notation with *symbolic* Kripke structures, in which the initial set and the transition function are represented by propositional formulae over state variables. We assume that the set of state variables is the same as the set of atomic propositions and hence eliminate the labelling function.

LTL semantics Section 2.5. The semantics are given in terms of first-order logic with an explicitly reduced syntax to simplify the encoding given in the following chapter. The LTL semantics are defined over paths through symbolic Kripke structures, rather than more general linear structures.

Chapter 3

Bounded Model Checking

Bounded model checking (BMC) [12] is a technique for symbolic model checking (Section 2.6) which replaces the BDD data structure, and its associated space-explosion problems, with Boolean SAT solver technology (see Section 2.2). This involves a number of restrictions to the original problem. Whilst symbolic model checking, through the BDD representation of sets of states, naturally handles specifications in CTL, BMC represents a single path and hence naturally captures LTL. In order to make the problem manageable for SAT, only a finite number of states are represented. In symbolic model checking the transition relation is applied on demand during solving; in BMC a bounded number of iterations of the transition relation is passed directly to the SAT solver.

In this chapter we give a new presentation of the BMC approach of Biere, Cimatti, Clarke, and Zhu [12]. Although we arrive at the same encoding as Biere et al., the route we take and its justification are radically different. By deriving bounded model checking from infinite-time LTL model checking we make clear the tradeoffs inherent in the approach. This method also assists us with the further development of the encoding in Chapter 5.

3.1 Infinite and Finite Paths

As noted in Section 2.6, we consider the existential LTL model checking problem: find a witness to the existential $\exists \pi \in M \;.\; \pi \models \phi$; this is equivalent to finding a counterexample to the universal problem $\forall \pi \in M \;.\; \pi \models \neg\phi$. Typically, the LTL formula is phrased in the universal sense, as a property

which is expected to hold in a correct model, and the problem is rephrased as an existential one by negating the LTL formula.

The traditional semantics of LTL is over a path of infinite length, although the witness may take only a finite number of states to establish (this depends on the nature of the specification). Bounded model checking takes advantage of this distinction by attempting to find a representation of the witness considering only a bounded number of transitions, k, hence $k + 1$ states. In Biere et al. [12], these states are interpreted in two different ways, described below, which correspond roughly to the finite and infinite witnesses.

3.1.1 k-Prefix Paths

Some LTL properties, such as **F** and **R**, are satisfied by a property holding in a single reachable state. For example, all witnesses to the property **F** p must pass through a state in which p holds. It is sufficient to identify the finite sequence of states leading to such a state; the evolution of the system after this point is irrelevant. For BMC, if a satisfying state occurs within k transitions from an initial state then a sequence of $k + 1$ states may be given as the bounded witness.

In BMC, we consider directly a finite path of $k + 1$ states; this means that we must give a new semantics of LTL over finite paths[1]. We begin by defining the path prefix operator which allows us to relate infinite paths to their finite prefixes.

Definition 3.1.1 (k-prefix)

The k-prefix of a path π is the path consisting of the first $k + 1$ states in π, and is written $\pi|_k$.

Finite paths are defined in a similar way to infinite paths; here we give a definition of finite paths through a Kripke structure in a similar manner to Definition 2.4.5.

Definition 3.1.2 (finite path)

A finite path[2] ϖ through a Kripke structure $M = \langle S, T, L, I \rangle$ is a sequence

[1] This is slightly different to the approach taken by Biere et al. [12], who define a finite semantics over infinite paths. We believe that our approach is more flexible and makes the origin of the sound and complete semantics more apparent.

[2] The symbol ϖ, LaTeX's "variant π" is used to distinguish finite paths from infinite ones. See also page xvi.

of states $s_0, s_1, \ldots, s_n \in S$ such that $s_0 \in I$ and for all $i \in \mathbb{N}$, $\langle s_i, s_{i+1} \rangle \in R$. We write the length of the path as $|\varpi| = n + 1$.

We refine the LTL model checking problem to focus on finite prefixes by splitting the formula into two parts: the first $k + 1$ states and the infinite suffix. Identifying a finite path ϖ with a finite prefix of an infinite path $\pi|_k$ reveals the following equation

$$\exists \pi \in M \,.\, \phi \quad \equiv \quad \exists \varpi \in M, |\varpi| = k + 1 \,.\, \exists \pi \in M, \pi|_k = \varpi \,.\, \pi \models \phi$$

We perform bounded model checking of finite prefixes in terms of the first existential ($\exists \varpi \in M, |\varpi| = k + 1$) given above. In this chapter we develop bounded model checking as a semantics of the finite prefix paths and relate it to the infinite formula.

Definition 3.1.3 (k-prefix BMC)
For a finite path ϖ and LTL formula ϕ, let the finite prefix bounded interpretation of ϕ starting from the ith state be $\varpi \models_k^i \phi$. We write $\models_k$ for $\models_k^0$. Definitions of $\models_k^i$ are given in the following sections.

The finite prefix bounded model checking problem is to decide the truth of

$$\exists \varpi \in M, |\varpi| = k + 1 \,.\, \varpi \models_k \phi$$

The key idea is that the $\models_k^i$ relation is defined without the ability to refer to states after k. The assumptions made about the evolution of the system after the kth state determine the soundness or completeness of the BMC procedure, and form part of the definition of the $\models_k^i$ relation.

3.1.1.1 Sound Semantics of LTL on k-Prefix Paths

A sound BMC procedure over k-prefixes produces a witness only when it can be guaranteed that such a witness is a prefix of a witness in the full semantics. Formally,

$$\forall \phi, \forall M, \forall \varpi \in M, |\varpi| = k + 1 \,.\, (\varpi \models_k \phi) \rightarrow (\forall \pi \in M, \pi|_k = \varpi \,.\, \pi \models \phi)$$

This means that bugs are identified by the procedure, but not the absence of bugs. This is the behaviour given by Biere et al. [12].

It would be easy to give a semantics which is sufficient to satisfy the above condition, but would be useless in practice. The challenge of choosing a sound semantics is to provide an interpretation which is sufficiently complete to be useful, without sacrificing the soundness condition. In fact, we can conceive of the partial order of bounded semantics operators $\langle \models_k, \sqsubseteq \rangle$ with respect to completeness:

$$\models_k \sqsubseteq \models'_k \quad \Leftrightarrow \quad \forall \phi, \forall M, \forall \varpi \in M, |\varpi| = k + 1 \;.\; (\varpi \models_k f) \rightarrow (\varpi \models'_k \phi)$$

We are interested in the sound relations which are maxima of this partial order. To completely address this issue is beyond the scope of this thesis (see Section 10.2.1) and we instead give an informal explanation for the derivation of the semantics given below.

We derive a bounded semantics from the infinite case in Figure 2.8 by modifying each case to take into account the finiteness of the path. Consider, for example, the **F** operator. $\mathbf{F}\,\phi$ is satisfied if a state exists in which ϕ holds; it is violated if no such state exists. Examining a finite number of states, we can be sure that ϕ holds in some state if it is seen in a state before the bound; we cannot be similarly sure that it is violated if no such state is found as the occurrence may be after the bound (see Figure 3.1). We thus derive the semantics

$$\varpi \models^i_k \mathbf{F}\,\phi \quad \Leftrightarrow \quad \begin{cases} \exists j, i \leq j \leq k \;.\; \varpi \models^j_k \phi & \text{if } \mathbf{F}\,\phi \text{ positive polarity} \\ \bot & \text{if } \mathbf{F}\,\phi \text{ negative polarity} \end{cases}$$

We can make similar reasoning about the **G** operator, and derive the semantics

$$\varpi \models^i_k \mathbf{G}\,\phi \quad \Leftrightarrow \quad \begin{cases} \bot & \text{if } \mathbf{G}\,\phi \text{ positive polarity} \\ \exists j, i \leq j \leq k \;.\; \varpi \models^j_k \phi & \text{if } \mathbf{G}\,\phi \text{ negative polarity} \end{cases}$$

To obtain a semantics that is not dependent on polarity it is sufficient to require the LTL formula to be in NNF (see Figure 2.9), and hence every operator to appear only with positive polarity. In this case, for the sound k-prefix semantics, **F** and **G** are no longer logical duals; this also applies to the other duals usually seen in LTL: **U** and **R**, and the self-dual **X**.

The other operators can be derived similarly to **G** and **F** given above, although the **R** operator requires a little more explanation. The expression given in Figure 2.8 can be rearranged to a form more readily restricted to a finite path (this originates in Biere et al. [12]).

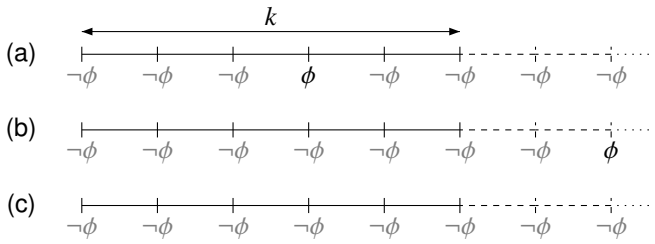


Figure 3.1: Three finite cases for $\mathbf{F}\,\phi$: in (a) ϕ occurs before k; in (b) it occurs after k while in (c) it does not occur at all. (b) and (c) are indistinguishable when examining the first k states

Lemma 3.1 (simplification of R) *For all LTL formulae ϕ and ψ,*

$$\phi\,\mathbf{R}\,\psi \equiv \mathbf{G}\,\psi \vee \psi\,\mathbf{U}(\phi \wedge \psi)$$

Proof From the semantics, we obtain

$$\begin{aligned} &\forall n, i \leq n \wedge (\forall j, i \leq j < n \;.\; \pi \not\models^j \phi) \;.\; \pi^n \models \psi \\ &\quad\equiv \quad (\forall m, i \leq m \;.\; \pi \models^m \psi) \vee \exists m, i \leq m \;.\; \pi \models^m \phi \wedge \forall l, i \leq l \leq m \;.\; \pi \models^l \psi \end{aligned}$$

If ψ holds in every state after i then the two sides are trivially equivalent.

Otherwise, for $\Rightarrow$, consider the smallest m such that $\pi^m \models \psi$; then for every $l < m$ we have $\pi \not\models^l \phi$ and hence from the left hand side for every $l \leq m$ we have $\pi \models^l \psi$.

For $\Leftarrow$, consider $n > m$ — $\exists j < n \;.\; \pi \models^j \phi$ holds for $j = m$ making the left hand side trivially true; for $n \leq m$, we have $\forall i \leq n \leq m \;.\; \pi \models^n \psi$ from the right hand side expression. □

In the resulting bounded semantics, the **G** is eliminated as discussed above; the remaining expression has an existential outermost so can be directly restricted to $k + 1$ states. The full sound bounded semantics of LTL, assuming NNF, are given in Figure 3.2, and are the same as those given by Biere et al. [12]. They are believed to be maximally complete for the case where semantics are restricted to definitions of single operators.

3.1.1.2 Complete Semantics of LTL on k-Prefix Paths

While not mentioned in the literature, a full exposition should include a discussion of the alternative semantics. A complete BMC procedure over k-prefixes

$$
\begin{array}{lcl}
\varpi \models_k^i a & \Leftrightarrow & a \in \varpi(i) \\
\varpi \models_k^i \neg a & \Leftrightarrow & a \notin \varpi(i) \\
\varpi \models_k^i \phi \wedge \psi & \Leftrightarrow & (\varpi \models_k^i \phi) \wedge (\varpi \models_k^i \psi) \\
\varpi \models_k^i \phi \vee \psi & \Leftrightarrow & (\varpi \models_k^i \phi) \vee (\varpi \models_k^i \psi) \\
\varpi \models_k^i \mathbf{X}\phi & \Leftrightarrow & \begin{cases} \varpi \models_k^{i+1} \phi & \text{if } i < k \\ \bot & \text{otherwise} \end{cases} \\
\varpi \models_k^i \mathbf{F}\phi & \Leftrightarrow & \exists j, i \leq j \leq k \;.\; \varpi \models_k^j \phi \\
\varpi \models_k^i \mathbf{G}\phi & \Leftrightarrow & \bot \\
\varpi \models_k^i \phi \,\mathbf{U}\, \psi & \Leftrightarrow & \exists j, i \leq j \leq k \;.\; \left(\varpi \models_k^j \psi \wedge \forall n, i \leq n < j \;.\; \varpi \models_k^n \phi\right) \\
\varpi \models_k^i \phi \,\mathbf{R}\, \psi & \Leftrightarrow & \exists j, i \leq j \leq k \;.\; \left(\varpi \models_k^j \phi \wedge \forall n, i \leq n \leq j \;.\; \varpi \models_k^j \psi\right)
\end{array}
$$

Figure 3.2: The sound semantics of NNF LTL for k-prefix paths. The model M is implicit, with $\varpi \in M$ and $|\varpi| = k + 1$

fails to produce a witness only when it can be guaranteed that a witness is not available in the full semantics. That is,

$$\forall \phi, \forall M, \forall \varpi \in M, |\varpi| = k + 1 \;.\; (\exists \pi \in M, \pi|_k = \varpi \;.\; \pi \models \phi) \rightarrow (\varpi \models_k \phi)$$

This means that the procedure identifies all correct models, but not all buggy models. As before, we can define a partial order of bounded semantics operators $\langle \models_k, \sqsubseteq' \rangle$ with respect to soundness:

$$\models_k \sqsubseteq' \models'_k \quad \Leftrightarrow \quad \forall \phi, \forall M, \forall \varpi \in M, |\varpi| = k + 1 \;.\; (\varpi \models'_k \phi) \rightarrow (\varpi \models_k \phi)$$

We take the same approach as above to derive sound, complete bounded semantics of NNF LTL, given in Figure 3.3. Notice that in this case, the semantics for **R** is close to the infinite semantics, but we have transformed the semantics for **U** by referring to the dual of Lemma 3.1, $\phi \,\mathbf{U}\, \psi \equiv \mathbf{F}\,\psi \wedge \psi \,\mathbf{R}(\phi \vee \psi)$.

3.1.2 k-Loop Paths

Finding witnesses to properties such as **G** and **U** requires a demonstration of a property holding forever: a witness of infinite length. For example, all

$$
\begin{array}{lcl}
\varpi \models_k^i a & \Leftrightarrow & a \in \varpi(i) \\
\varpi \models_k^i \neg a & \Leftrightarrow & a \notin \varpi(i) \\
\varpi \models_k^i \phi \wedge \psi & \Leftrightarrow & (\varpi \models_k^i \phi) \wedge (\varpi \models_k^i \psi) \\
\varpi \models_k^i \phi \vee \psi & \Leftrightarrow & (\varpi \models_k^i \phi) \vee (\varpi \models_k^i \psi) \\
\varpi \models_k^i \mathbf{X}\,\phi & \Leftrightarrow & \begin{cases} \varpi \models_k^{i+1} \phi & \text{if } i < k \\ \top & \text{if } i \geq k \end{cases} \\
\varpi \models_k^i \mathbf{F}\,\phi & \Leftrightarrow & \top \\
\varpi \models_k^i \mathbf{G}\,\phi & \Leftrightarrow & \forall j, i \leq j \leq k \;.\; \varpi \models_k^j \phi \\
\varpi \models_k^i \phi\,\mathbf{U}\,\psi & \Leftrightarrow & \forall j, i \leq j \leq k \;.\; \left(\varpi \models_k^j \phi \vee \exists n, i \leq n \leq j \;.\; \varpi \models_k^j \psi\right) \\
\varpi \models_k^i \phi\,\mathbf{R}\,\psi & \Leftrightarrow & \forall j, i \leq j \leq k \;.\; \left(\varpi \models_k^j \psi \vee \exists n, i \leq n < j \;.\; \varpi \models_k^j \phi\right)
\end{array}
$$

Figure 3.3: The complete semantics of NNF LTL for k-prefix paths. The model M is implicit, with $\varpi \in M$ and $|\varpi| = k + 1$

witnesses to $\mathbf{G}\,p$ must show p holding in every state.

In the unbounded domain, the common approach to LTL model checking (see Gerth et al. [56]) is by conversion of the LTL formula to an equivalent Büchi automaton, and searching for an accepting path in the product automaton formed with the model. By the definition of Büchi automata (Section 2.4.1) this means that the path eventually reaches a cycle in the strongly connected component containing the acceptance state. Without loss of generality, we can restrict this path to be of the form $\pi = ab^\omega$—an infinitely repeating sequence of states following some finite length prefix.

The key observation made by Biere et al. [12] is that a path of the form ab^ω is representable in a finite number of states $k + 1 = |ab|$ together with a record of the length of the prefix component $l = |a|$. We call such a path a *k-l*-loop path. We define these terms formally below, and illustrate them in Figure 3.4.

Definition 3.1.4 (*k*-loop)

We say the a path π is a *k-loop* if for all $i \geq 0$, $\pi(k + i) \equiv \pi(l + i)$ for some l, $0 \leq l < k$. We call l the *loopback point*. A k-loop with a loopback point l is referred to as a *k-l-loop*. The *period* of the loop part of a *k-l*-loop path is $p = k - l$.

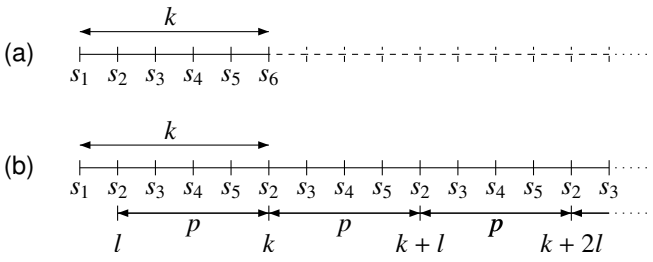


Figure 3.4: Graphical depiction of k, l, and p given in Definition 3.1.4. (a) shows the prefix path case, dashes indicating the unconsidered states; (b) shows the loop path case

Let the set $\mathsf{loop}_{k,l}$ be the set of all k-l-loop paths and loop_k be the set of all k-loop paths. The set of all loop paths is $\mathsf{loop} = \bigcup_{k \in \mathbb{N}} \mathsf{loop}_k$.

Unlike the k-prefix paths, where every infinite path has a corresponding k-prefix path, the set of k-loop paths are a subset of the infinite paths. That is, not every infinite path is, or can be represented as, a k-loop path. However, since k-loop paths are infinite paths, we can use the usual semantics of LTL unchanged.

We formalise the idea of k-loop BMC by defining the $\models^{\circ}$ operator over finite paths in terms of general LTL model checking.

Definition 3.1.5 (k-loop BMC)
We define the k-loop model checking operator $\models^{\circ}$ for a given loopback point l as

$$\forall \phi, \forall M, \forall \varpi \in M \;.\; \varpi \models^{\circ}_{k,l} \phi \Leftrightarrow \exists \pi \in M, \pi|_k = \varpi \;.\; \pi \in \mathsf{loop}_{k,l} \wedge \pi \models \phi$$

In practice, we wish to restrict our attention explicitly to the first k states of the path. This is achieved by projecting later states onto their canonical representation within the first k states. The projection function, $\rho_0(i,k,l)$, defined by Benedetti and Cimatti [8], is a function over path indices: the ith state is mapped onto its corresponding state in the loop if it is beyond the kth state but left in place otherwise[3].

$$\rho_0(i,k,l) = \begin{cases} i & i < k \\ i - k + l & \text{otherwise} \end{cases}$$

[3]Benedetti and Cimatti [8] define ρ_n to enable a projection to the nth iteration of the loop as required for their treatment of past time LTL; for the pure future treatment here, ρ_0 is sufficient.

We transform the infinite semantics of LTL over k-loop paths by applying ρ_0 to every state index. By a series of case splits we can arrive at the specialised semantics of LTL for k-loop paths, given in full in Figure 3.5, which refers only to the first k states of a k-loop path. For example, the semantics for $\mathbf{G}\,\phi$ becomes

$$\begin{aligned}\forall j, i \leq j \;.\; \pi \models^{\rho(i,k,l)} \phi \equiv\; & (\forall j, i \leq j < k \;.\; \pi \models^{i} \phi) \\ & \wedge (\forall j, i \leq k \leq j < 2k - l \;.\; \pi \models^{i-k-l} \phi) \\ & \wedge (\forall j, i \leq 2k - l \leq j < 3k - 2l \;.\; \pi \models^{i-2k-2l} \phi) \\ & \wedge \ldots \\ \equiv\; & (\forall j, i \leq j < l \;.\; \pi \models^{i} \phi) \wedge (\forall j, l \leq j < k \;.\; \pi \models^{i} \phi) \\ \equiv\; & (\forall j, \min(i, l) \leq j < k \;.\; \pi \models^{i} \phi)\end{aligned}$$

The correctness of this semantics follows from

$$\forall \pi \in \mathsf{loop}_{k,l} \;.\; \pi \models^{i} \phi \Leftrightarrow \pi \models^{\rho_0(i,k,l)} \phi$$

which holds for all (future time) LTL ϕ because the semantics is defined in terms of the states $\pi(j)$ for $j \geq i$; the loop property means that $\pi(j) = \pi(\rho_0(j, k, l))$ for all $j \geq i$.

3.1.3 Combined BMC

Both the k-prefix and the k-loop style BMC operate over a sequence of k transitions through the model. The two methods can be performed simultaneously by noting that, given a finite path $\varpi \in M$ of length k,

$$\exists l < k \;.\; \varpi(k) = \varpi(l) \rightarrow \exists \pi \in \mathsf{loop}_{k,l} \;.\; \pi|_k = \varpi$$

We write down the expression of the combined bounded model checking problem in two parts to simplify the correctness proof later on.

Definition 3.1.6 (combined bounded model checking problem)
The *(sound) (combined) bounded model checking problem with bound* k for a model M and an LTL formula ϕ is to determine whether a finite prefix path of length k or a k-loop path exists in M which satisfies ϕ:

$$\mathrm{BMC}(M, \phi, k) \doteq \exists \varpi \in M, |\varpi| = k + 1 \;.\; \mathrm{BMC}(\phi, k, \varpi)$$

$$\mathrm{BMC}(\phi, k, \varpi) \doteq (\varpi \models_k \phi \vee (\exists l < k \;.\; \varpi(k) = \varpi(l) \wedge \varpi \models^{\circ}_{k,l} \phi))$$

$$
\begin{array}{lcl}
\varpi \models^{i}_{k,l} a & \Leftrightarrow & a \in \varpi(i) \\
\varpi \models^{i}_{k,l} \neg\phi & \Leftrightarrow & \varpi \not\models^{i}_{k,l} \phi \\
\varpi \models^{i}_{k,l} \phi \wedge \psi & \Leftrightarrow & (\varpi \models^{i}_{k,l} \phi) \wedge (\varpi \models^{i}_{k,l} \psi) \\
\varpi \models^{i}_{k,l} \phi \vee \psi & \Leftrightarrow & (\varpi \models^{i}_{k,l} \phi) \vee (\varpi \models^{i}_{k,l} \psi) \\
\varpi \models^{i}_{k,l} \mathbf{X}\phi & \Leftrightarrow & \varpi \models^{\rho_0(i+1)}_{k,l} \phi \\
\varpi \models^{i}_{k,l} \mathbf{F}\phi & \Leftrightarrow & \exists j, \min(i,l) \leq j < k \;.\; \varpi \models^{j}_{k,l} \phi \\
\varpi \models^{i}_{k,l} \mathbf{G}\phi & \Leftrightarrow & \forall j, \min(i,l) \leq j < k \;.\; \varpi \models^{j}_{k,l} \phi \\
\varpi \models^{i}_{k,l} \phi \mathbf{U} \psi & \Leftrightarrow & (\exists j, i \leq j < k \;.\; \varpi \models^{j}_{k,l} \psi \wedge \forall n, i \leq n < j \;.\; \varpi \models^{n}_{k,l} \phi) \\
 & & \vee (\exists j, l \leq j < i \;.\; \varpi \models^{j}_{k,l} \psi \wedge \\
 & & \quad \forall n, i \leq n < k \;.\; \varpi \models^{n}_{k,l} \phi \wedge \forall n, l \leq n < j \;.\; \varpi \models^{n}_{k,l} \phi) \\
\varpi \models^{i}_{k,l} \phi \mathbf{R} \psi & \Leftrightarrow & (\forall j, i \leq j < k \;.\; \varpi \models^{j}_{k,l} \psi \vee \exists n, i \leq n < j \;.\; \varpi \models^{n}_{k,l} \phi) \\
 & & \wedge (\forall j, l \leq j < i \;.\; \varpi \models^{j}_{k,l} \psi \vee \\
 & & \quad \exists n, i \leq n < k \;.\; \varpi \models^{n}_{k,l} \phi \vee \exists n, l \leq n < j \;.\; \varpi \models^{n}_{k,l} \phi)
\end{array}
$$

Figure 3.5: The projected semantics of LTL for k-loop paths

The soundness of the formula in this definition with respect to infinite-path model checking follows trivially from the soundness of the two components. We can, of course, make the same extension for the complete case, giving the combined formula

$$\exists \varpi \in M, |\varpi| = k + 1 \;.\; (\varpi \models_k \phi \wedge (\exists l < k \;.\; \varpi(k) = \varpi(l) \rightarrow \varpi \stackrel{\circ}{\models}_{k,l} \phi)).$$

From this point onwards, we focus on the sound BMC procedure as it more closely fits the usual usage pattern of model checking: finding bugs in systems.

3.1.4 Witness Length

From the discussion above, we can see that the type of witness generated by BMC can take two forms, depending on the type of property:

- Properties with infinite witness: a loop of size $k - l$ with a path of length l connecting it to the set of initial states (Figure 3.6a).
- Properties with finite witness: either a k-bounded path (Figure 3.6b) or a k-loop (Figure 3.6c).

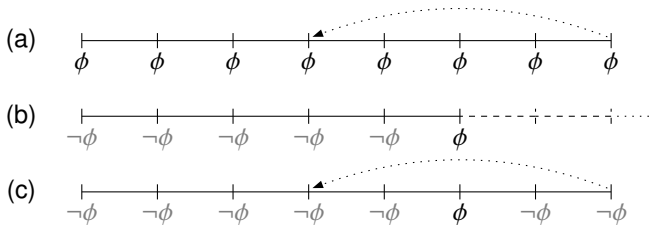


Figure 3.6: Path examples: (a) k-loop as an infinite witness; (b) k-bounded path as a finite witness; (c) k-loop as a finite witness

Allowing the witness to a finite property to be a k-loop changes the behaviour of BMC: rather than the shortest witness being the path with the fewest states, it is the path with the most compact representation.

For example, consider the modulo-n counter shown in Figure 3.7a, with the specification $\mathbf{F}(x = n \wedge \mathbf{F}(x = n - 1 \wedge \mathbf{F}(x = n - 2 \wedge \cdots)))$ which checks that the values may be seen in reverse order in the behaviour of the counter. The shortest witness to this is a path of length $n(n-1)$ as each successive decreasing value is separated from the next by $n - 1$ intermediate states (Figure 3.7b). The

most compact representation of this, however, is a path of length n with a loop from the nth state to the 1st (Figure 3.7c), as this covers the n iterations of the counter.

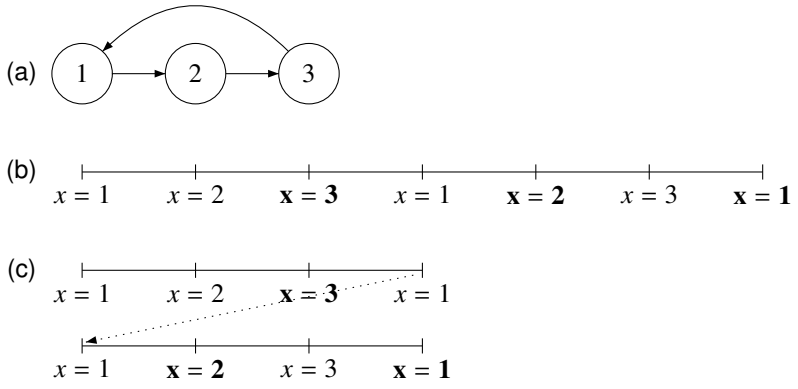


Figure 3.7: Counter example, $n = 3$

This ability to find short witnesses is one of the main strengths of BMC. As it is difficult (NP-hard) to find the value of k which makes BMC equivalent to model checking in general, we run BMC at increasing k until a witness is found. The result is thus the witness with the most compact representation. As this is a powerful restriction on the amount of state space considered, the performance gain over symbolic model checking can be considerable; on the other hand, it becomes difficult to prove that a specification is correct, as k must be increased until all potential witness have been eliminated. In the worst case this is at the diameter (the length of the longest shortest path) of the model.

3.2 The Bounded Model Checking Encoding

As propositional satisfiability (see Section 2.2) is an existential procedure, the BMC problem encodes naturally to a SAT problem. We follow Biere et al. [12] in using the notation $[\![\ldots]\!]$ to indicate encoding to propositional logic, writing ${}_l[\![\zeta]\!]^i_k$ to indicate the encoding of ζ in state i with bound k and loopback point l and writing $l = -$ to indicate the finite prefix encoding: ${}_-[\![\zeta]\!]^i_k$.

Rewriting Definition 3.1.6 with $\varpi \in M$ moved from the range of the quantification to the argument ('trading'), we obtain

$$\exists \varpi, |\varpi| = k + 1 \;.\; \varpi \in M \wedge (\varpi \models_k f \vee (\exists l < k \;.\; \varpi(k) = \varpi(l) \wedge \varpi \mathrel{\mathring{\models}}_{l,k} f))$$

The propositional encoding is obtained by rewriting the quantifiers in the QTPL expressions as propositional connectives over ranges (existentials becoming disjunctions, universals becoming conjunctions) resulting in the formula given below, and by introducing a set of atomic propositions to correspond to the variables of the Kripke transition relation.

$$\mathrm{BMC}_{\mathrm{p}}(\hat{M}, \phi, k) \doteq [\![\hat{M}]\!]_k \wedge \left({}_{-}[\![\phi]\!]_k^0 \vee \bigvee_{l<k} \left({}_{l}[\![\varpi(k) = \varpi(l)]\!]_k^0 \wedge {}_{l}[\![\phi]\!]_k^0 \right) \right)$$

3.2.1 States and the Model

$\mathrm{BMC}(\hat{M}, \phi, k)$ is defined in terms of the symbolic Kripke structure $\hat{M} = \langle A, \hat{I}, \hat{T} \rangle$. We therefore use $k + 1$ copies of the set A of the atomic propositions used in the Kripke structure and the LTL property ϕ, written A^i for $0 \leq i \leq k$. Each copy is used to represent a state on the path, so the state $\varpi(i)$ corresponds to the truth assignment to the atomic propositions in A^i.

The encoding of the model follows directly from the definition of symbolic Kripke structures (Definition 2.4.3):

$$[\![\hat{M}]\!]_k \quad \Leftrightarrow \quad \hat{I}(A^0) \wedge \bigwedge_{0 \leq i < k} \hat{T}(A^i, A^{i+1})$$

For symbolic Kripke structures, the truth assignment for the propositions representing a state is sufficient to completely identify that state. Testing that two states are the same, therefore, is done by testing the equivalence of each proposition:

$${}_{l}[\![\varpi(k) = \varpi(l)]\!]_k \quad \Leftrightarrow \quad \bigwedge_{a \in A} a^k \Leftrightarrow a^l$$

3.2.2 k-Prefix Paths

Consider the first disjunct from $\mathrm{BMC}_{\mathrm{p}}(\hat{M}, \phi, k)$,

$${}_{-}[\![\phi]\!]_k^0$$

The semantics in Figure 3.2 relate ϕ over a k-prefix path to a QTPL expression. As the quantifiers used are over linear restrictions of natural number variables, we replace quantifiers with conjunctions or disjunctions, the variables and linear restrictions moving to the metalanguage. This results in the recursive encoding of LTL given in Table 3.1.

Table 3.1: The BMC encoding for LTL, prefix path case

ϕ	$[\![\phi]\!]_k^i$
a	a^i
$\neg a$	$\neg a^i$
$\phi \wedge \psi$	$[\![\phi]\!]_k^i \wedge [\![\psi]\!]_k^i$
$\phi \vee \psi$	$[\![\phi]\!]_k^i \vee [\![\psi]\!]_k^i$
$\mathbf{X}\,\phi$	$i < k \wedge [\![\phi]\!]_k^{i+1}$
$\mathbf{F}\,\phi$	$\bigvee_{j=i}^{k} [\![\phi]\!]_k^j$
$\mathbf{G}\,\phi$	$\bot$
$\phi\,\mathbf{U}\,\psi$	$\bigvee_{j=i}^{k} \left([\![\psi]\!]_k^j \wedge \bigwedge_{n=i}^{j-1} [\![\phi]\!]_k^n \right)$
$\phi\,\mathbf{R}\,\psi$	$\bigvee_{j=i}^{k} \left([\![\phi]\!]_k^j \wedge \bigwedge_{n=i}^{j} [\![\psi]\!]_k^n \right)$

3.2.3 k-Loop Paths

The second disjunction includes the loopback condition ${}_l[\![\varpi(k) = \varpi(l)]\!]_k^0$ and the encoding of the LTL property over k-loop paths:

$${}_l[\![\phi]\!]_k^0$$

The loopback condition is an instance of the identification of two states, defined above. As before, we notice that the semantics given in Figure 3.5 can used to generate a propositional encoding by replacing quantifiers with conjunctions and disjunctions and reinterpreting the linear restrictions of variables as part of the metalanguage. The resulting recursive encoding of LTL is given in Table 3.2.

3.2.4 Correctness

We show that the BMC encoding is a correct propositional representation of the combined sound BMC formula given in Definition 3.1.6. The relationship between BMC and infinite state model checking then follows from the

Table 3.2: The BMC encoding for LTL, loop path case

ϕ	${}_l[\![\phi]\!]^i_k$
a	a^i
$\neg a$	$\neg a^i$
$\phi \wedge \psi$	${}_l[\![\phi]\!]^i_k \wedge {}_l[\![\psi]\!]^i_k$
$\phi \vee \psi$	${}_l[\![\phi]\!]^i_k \vee {}_l[\![\psi]\!]^i_k$
$\mathbf{X}\,\phi$	$\left(i < k \wedge {}_l[\![\phi]\!]^{i+1}_k\right) \vee \left(i = k \wedge {}_l[\![\phi]\!]^l_k\right)$
$\mathbf{F}\,\phi$	$\bigvee_{j=\min(i,l)}^{k} {}_l[\![\phi]\!]^j_k$
$\mathbf{G}\,\phi$	$\bigwedge_{j=\min(i,l)}^{k} {}_l[\![\phi]\!]^j_k$
$\phi\,\mathbf{U}\,\psi$	$\bigvee_{j=i}^{k}\left({}_l[\![\psi]\!]^j_k \wedge \bigwedge_{n=i}^{j-1} {}_l[\![\phi]\!]^n_k\right) \vee \bigvee_{j=l}^{i-1}\left({}_l[\![\psi]\!]^j_k \wedge \bigwedge_{n=i}^{k} {}_l[\![\phi]\!]^n_k \wedge \bigwedge_{n=l}^{j-1} {}_l[\![\phi]\!]^n_k\right)$
$\phi\,\mathbf{R}\,\psi$	$\bigwedge_{j=i}^{k}\left({}_l[\![\psi]\!]^j_k \vee \bigvee_{n=i}^{j-1} {}_l[\![\phi]\!]^n_k\right) \wedge \bigwedge_{j=l}^{i-1}\left({}_l[\![\psi]\!]^j_k \vee \bigvee_{n=i}^{k} {}_l[\![\phi]\!]^n_k \vee \bigvee_{n=l}^{j-1} {}_l[\![\phi]\!]^n_k\right)$

soundness results given earlier in this chapter.

Theorem 3.2 (correctness of BMC encoding) *$\sigma \models \mathrm{BMC}_\mathrm{p}(\hat{M}, \phi, k)$ if and only if σ represents a path $\varpi \in K(\hat{M})$ which satisfies* $\mathrm{BMC}(\phi, k, \varpi)$.

Proof The set of propositions in $\mathrm{BMC}_\mathrm{p}(\hat{M}, \phi, k)$ is $\bigcup_{0\le i\le k} A^i$ where A is the set of atomic propositions in the symbolic Kripke structure $\hat{M}$. Writing $\lhd$ for the domain restriction operator, we can write the path represented by σ as $\varpi = A^0 \lhd \sigma, A^1 \lhd \sigma, \ldots A^k \lhd \sigma$.

The proof now breaks into four parts corresponding to the four parts of $\mathrm{BMC}_\mathrm{p}(\hat{M}, \phi, k)$: we show that the solutions σ coresponds to the valid bounded paths in $K(\hat{M})$; that the solutions which satisfy the prefix encoding correspond to the paths which satisfy the prefix semantics; that the solutions which satisfy the loop property correspond to the k-loop paths; and that the solutions which satisfy the k-loop encoding correspond to the paths which satisfy the k-loop semantics.

1. By appealing to the definition of $\mathrm{BMC}_{\mathrm{p}}(\hat{M}, \phi, k)$ and the definition of $K(\hat{M})$ in Section 2.4.2, we see that $\mathrm{BMC}_{\mathrm{p}}(\hat{M}, \phi, k)$ constrains A^0 as strongly as $\varpi \in K(\hat{M})$ constrains $\varpi(0)$; similarly, $\mathrm{BMC}_{\mathrm{p}}(\hat{M}, \phi, k)$ constrains A^i and A^{i+1} as strongly as $\varpi \in K(\hat{M})$ constrains $\varpi(i)$ and $\varpi(i+1)$.

2. The encoding given in Table 3.1 represents precisely the same constraint on propositions $\bigcup_{0 \leq i \leq k} A^i$ as the semantics in Figure 3.2 does on the path $\varpi = A^0 \lhd \sigma, A^1 \lhd \sigma, \ldots A^k \lhd \sigma$. This can be shown trivially by induction, with base cases of the atomic propositions and their negations; since the range of every quantifier is finite they are equivalent to the conjuctions and disjunctions in the encoding.

3. The constraint equating the propositions in A^l and A^k is equivalent to constraining ϖ to being a k-loop path by the observation at the start of Section 3.1.3.

4. The encoding given in Table 3.2 represents precisely the same constraint on propositions $\bigcup_{0 \leq i \leq k} A^i$ as the semantics in Figure 3.5 does on the path $\varpi = A^0 \lhd \sigma, A^1 \lhd \sigma, \ldots A^k \lhd \sigma$. As before, this can be shown trivially by induction. □

3.2.5 Encoding Fairness

We noted in Section 2.6.2 that fairness is a property in CTL*. However, it can also be expressed as the existential LTL property $\mathbf{G}\,\mathbf{F} \bigwedge_{F \in \mathcal{F}} \bigvee_{f \in F} f$. For symmetry with symbolic model checking, and for reasons of efficiency, it is sometimes considered separately and encoded directly to propositional logic. For each semantics of bounded model checking discussed above, we have

Sound semantics of k-prefix paths As before, the semantics of $\mathbf{G}$ collapses to $\bot$ giving the encoding of $\bot$ for all fair, sound k-prefix paths.

Complete semantics of k-prefix paths As before, the semantics of $\mathbf{F}$ collapses to $\top$ giving $\top$ for all fair, complete k-prefix paths.

k-loop paths We note that $\mathbf{G}\,\mathbf{F}\, f \rightarrow \mathbf{X}\,\mathbf{G}\,\mathbf{F}\, f$ as $\mathbf{G}\,\mathbf{F}\, f$ is both prefix closed and left-append closed[4]. This means that we can delay the evaluation of the

[4] *Prefix closed* means that if $\pi \models \phi$ then $\pi' \models \phi$ where π' is a suffix of π; *left-append closed* is the converse: if $\pi \models \phi$ then $\pi' \models \phi$ where π is a suffix of π'.

fairness condition so that only the states within the loop are considered. Furthermore, we can collapse the conjunction over states (corresponding to **G**) by noting that each conjunct is identical (by considering the projected semantics, Figure 3.5), giving the encoding

$$\bigvee_{l=0}^{k} \bigvee_{i=l}^{k} \bigwedge_{F \in \mathcal{F}} \bigvee_{s \in F} [\![s = \varpi(i)]\!]_k^i$$

Note that in the first case above, the semantics asserts that there are *no* fair k-prefix paths; we cannot therefore give a generalised encoding that covers BMC with and without fairness.

Cimatti et al. [23] give a similar improvement on the direct encoding of the LTL in the k-loop case by analysing the propositional logic produced; a further improvement is shown for the case where $|\mathcal{F}| = 1$, reducing this part of the encoding to linear size in k.

3.3 Encoding Approaches from the Literature

The encoding derived above differs only slightly from that given by Biere et al. [12]. Biere et al. explicitly restrict the prefix semantics to be evaluated only when the path exhibits no k-loop: that is, the k-loop encoding is guarded by a condition that for some l there exists a k-l-loop; Biere et al. give a similar guard for the prefix encoding, of the form

$$\neg \bigwedge_{l=0}^{k} {}_l[\![\varpi(l) = \varpi(k)]\!]_k$$

A more recent paper by Cimatti et al. [23] justifies the removal of this guard from the opposite direction: they show that for every loop path and every LTL formula, the existence of a witness from the prefix encoding implies the existence of one from the loop encoding (it is easy to see how this follows from our soundness arguments in Section 3.1.1.1), and hence the guard can be eliminated without changing the meaning of the formula. We note that the derivation above required no such justification.

Another significant difference between the encoding presented here and that given by Biere et al. [12] is in the definition of a loop path. In Definition 3.1.4 we give a loop path as one in which state $k + i$ and state $l + i$ are the same for

all $i \geq 0$, which leads to the assertion in the encoding that $\varpi(l) = \varpi(k)$. Biere et al. use instead the idea that there is a transition between $\varpi(k)$ and $\varpi(l)$. This means that $k + 1$ states capture $k + 1$ transitions, rather than k transitions in the present scheme. The disadvantage is that the encoding of a transition can be arbitrarily complex, while the encoding of an equality between states is known to be straightforward. Later work by the same authors, for example Cimatti et al. [23] and the NuSMV [22] implementation of BMC, uses the present definition of loop paths.

A significantly different approach to the encoding, proposed by de Moura, Rueß, and Sorea [36] and supported by the analysis of Clarke, Kroening, Ouaknine, and Strichman [28], is the use of classical LTL to Büchi automata conversion techniques, followed by a BMC implementation of a Büchi emptiness check on the product automaton formed with the model. We consider this technique in more detail in Section 8.

3.4 Complexity of Encodings

At first sight, the encoding described above is exponential in the size of the LTL: each of the infinite operators given in Tables 3.1 and 3.2 produces k or k^2 symbols multiplied by the size of the encoding of its arguments; an additional factor of k is required for the quantification over loopback points. For an LTL formula with a maximum nesting depth of n infinite-time operators, we produce $O(k^{2n+1})$ symbols in the final propositional logic.

Biere et al. [12] state that the encoding is polynomial in the size of the LTL formula and quadratic in k if all common subformulae are shared. The suggestion here, as put explicitly by Clarke, Kroening, Ouaknine, and Strichman [28], is that each instance of the specification encoding ${}_l[\![\phi]\!]^i_k$ can be renamed (see Section 2.2.1.1) to cause the sharing required. There are a maximum of $|\phi|k^2$ different instances, where $|\phi|$ is the number of subformulae of ϕ. This approach, directly applied to the transformations given in the chapter gives a complexity of $O(|f|k^4)$.

The reduction to quadratic size observed by Biere et al. is achieved by factorising the encodings of Tables 3.1 and 3.2 to obtain those given in Table 3.3. These are the encodings described explicitly in Biere et al., however it is not clear how the infinite recursion defined here can be related to the semantics.

Table 3.3: The factorised BMC Encoding for LTL

ϕ	$[\![\phi]\!]_k^i$	${}_l[\![\phi]\!]_k^i$
v	$v(i)$	$v(i)$
$\neg v$	$\neg v(i)$	$\neg v(i)$
$\phi \wedge \psi$	$[\![\phi]\!]_k^i \wedge [\![\psi]\!]_k^i$	${}_l[\![\phi]\!]_k^i \wedge {}_l[\![\psi]\!]_k^i$
$\phi \vee \psi$	$[\![\phi]\!] \vee [\![\psi]\!]$	${}_l[\![\phi]\!] \vee {}_l[\![\psi]\!]$
$\mathbf{X}\,\phi$	$[\![\phi]\!]_k^{i+1}$	${}_l[\![\phi]\!]_k^{\operatorname{succ} i}$
$\mathbf{F}\,\phi$	$[\![\phi]\!]_k^i \vee [\![\mathbf{F}\,\phi]\!]_k^{i+1}$	${}_l[\![\phi]\!]_k^i \vee {}_l[\![\mathbf{F}\,\phi]\!]_k^{\operatorname{succ} i}$
$\mathbf{G}\,\phi$	$\bot$	${}_l[\![\phi]\!]_k^i \wedge {}_l[\![\mathbf{G}\,\phi]\!]_k^{\operatorname{succ} i}$
$\phi\,\mathbf{U}\,\psi$	$[\![\psi]\!]_k^i \vee \left([\![\phi]\!]_k^i \wedge [\![\phi\,\mathbf{U}\,\psi]\!]_k^{i+1}\right)$	${}_l[\![\psi]\!]_k^i \vee \left({}_l[\![\phi]\!]_k^i \wedge {}_l[\![\phi\,\mathbf{U}\,\psi]\!]_k^{\operatorname{succ} i}\right)$
$\phi\,\mathbf{R}\,\psi$	$[\![\psi]\!]_k^i \wedge \left([\![\phi]\!]_k^i \vee [\![\phi\,\mathbf{R}\,\psi]\!]_k^{i+1}\right)$	${}_l[\![\psi]\!]_k^i \wedge \left({}_l[\![\phi]\!]_k^i \vee {}_l[\![\phi\,\mathbf{R}\,\psi]\!]_k^{\operatorname{succ} i}\right)$
ϕ	$\bot$ if $i > k$	

$$\operatorname{succ} i = \begin{cases} i+1 & \text{if } i < k, \text{ or} \\ l & \text{otherwise} \end{cases}$$

The implementation in NuSMV [22] does not follow these encodings directly, which suggests that they exist more as a device to justify the claim of quadratic size. On the other hand, these expressions are similar to the fixpoint expressions given in Section 2.5.4, and can be derived from them in a similar way to the original expressions.

3.5 Applications of BMC

The first big industrial example of verification using BMC is by Biere, Clarke, Raimi, and Zhu [13], who demonstrate the efficacy of the approach, although their example is limited to safety ($\mathbf{G}\,f$) properties.

Copty, Fix, Fraer, Giunchiglia, Kamhi, Tacchella, and Vardi [31] discuss the use of BMC at Intel on large, real-world designs (fragments of the Pentium

IV Processor) and compare an implementation of BMC using a SAT procedure with one using BDD-based satisfiability testing to give a clearer picture of the relative advantage of SAT. They find that in most cases the SAT-based verifier out-performs even a hand-tuned run of the BDD-based verifier. We note, however, that these results are difficult to repeat as both the verification software and the example circuits are internal to Intel.

For systems requiring detailed timing relationships, representing the model as a Kripke structure can be too restrictive or result in a blow-up in complexity. One solution is to use timed automata, which include constraints over the times at which transitions may be taken, together with a temporal logic augmented with linear constraints over time. Such systems can be efficiently handled by BMC if the SAT solver is extended to handle integer inequalities directly. Audemard, Cimatti, Kornilowicz, and Sebastiani [3] present a system built around their own SAT solver [4]. An alternative approach which is not as closely related to the work of Biere et al. [12] is that of Sorea [94] which is based on a conversion from LTL to Büchi automata, and so does not address the encoding directly.

Extensions to apply BMC to other temporal logics have been studied by Penczek, Woźna, and Zbrzezny, who have looked at the universal fragment of CTL [83], TCTL (in the timed automata domain) [82], and most recently ACTL* (the universal fragment of CTL*) [101]. As we observed at the start of this chapter, however, LTL is more closely suited to the BMC approach. Although these advances are interesting from a theoretical perspective, it is not clear that they hold any advantage over the alternatives of rewriting the specifications in LTL[5], or using a more suitable approach such as BDD-based model checking.

Benedetti and Cimatti [8] consider the extension of LTL to the past[6] by unrolling the transition relation extra times depending on a computed upper limit to the number of past states to be considered. This approach guarantees that the minimum length counterexample is obtained, at the expense of an encoding which is enlarged many times over the pure future case.

[5] Although there are expressions in LTL that cannot be written in CTL and vice versa, it is possible to write many common specifications in any of the usual temporal logics. Dwyer, Avruning, and Corbett [42] gives a wide range of useful patterns for specification in several temporal logics; in addition, Maidl [72] identifies theoretically the common subsets.

[6] While this does not increase the theoretical expressiveness of the logic, it can make certain specifications more succinct and easily understood.

3.6 Summary

This chapter gives a new presentation of the bounded model checking, including the encoding from LTL to propositional logic. The argument taken is as follows.

- Two interpretations of finite paths are possible: as prefixes of infinite paths, or as infinite loop paths in which the final $k - l$ states of the path are repeated.
- For each interpretation, we determine the way that LTL formulae are interpreted over these paths.
- We give two new modelling relations directly relating LTL formulae to finite paths.
- The BMC encoding is constructed by writing propositional formulae based directly on these finite semantics of LTL.
- Propositions in the encoding are copies (one for each state of the path) of the set of propositions in the symbolic Kripke structure.
- The model is encoded as constraints on these propositions by projecting the symbolic transition relation (which is already in propositional logic) to each pair of adjacent states.

This presentation also includes *complete* BMC semantics for LTL as the alternative to the usual sound approach. We observe that the sound and complete semantics given are the most straightforward, but not necessarily the best for the encoding and certainly not the only possible semantics. A heirarchy of semantics is proposed, but not explored further.

Chapter 4

The Separated Normal Form

The Separated Normal Form (SNF), a clausal normal form for temporal logic, is a concise way of representing temporal statements using a restricted syntax of only three temporal operators. SNF has its origins in executable temporal logic but also forms the basis of a clausal resolution proof system for temporal logic [46]. Although originally designed for a propositional linear time temporal logic, SNF and the resolution system have also been extended to CTL and first order temporal logic.

The first two sections of this chapter review existing SNF literature: we review the origins of SNF as described by Fisher for propositional temporal logic, and discuss the surrounding literature; we also describe Bolotov's variation on SNF for LTL [14].

The remaining sections detail the adaptations made to SNF for LTL in preparation for its use in BMC. We also address the issue of the transformation from LTL to SNF and its correctness in terms of the denotational semantics of LTL.

4.1 MetateM and SNF for PTL

The MetateM programming language[6] was created in the spirit of logic programming systems such as Prolog, but was constructed around temporal clause rules of the form

$$\text{past time antecedent} \rightarrow \text{future time consequent}$$

not unlike the Horn clause rule form of Prolog. The "past time antecedent" is a temporal logic statement about the past, and the "future time consequent" is a statement about the present or future, leading to a very natural "if this has happened, do this next" imperative interpretation. Given a set of temporal rules, each is checked at each time step, and those consequents following from true antecedents are executed. In terms of temporal logic, given a set of rules Ψ, the system checks $\mathbf{G}\left(\bigwedge_{\psi\in\Psi}\psi\right)$.

The division between future and past which is key to the imperative reading of MetateM was analysed separately by Gabbay [50] (who was also a co-author on the MetateM work). Gabbay observed that any temporal formula could be rewritten in terms of a propositional combination of future, present, and past components: a process called *separation*. Standard propositional techniques then enable us to obtain the form $\mathbf{G}\left(\bigwedge_i(P_i \Rightarrow F_i)\right)$ where P_i are past time formulae and F_i are future time formulae.

SNF and the language of MetateM take the semantic restrictions permitted by the separation theorem one step further, by limiting the set of temporal operators. Fisher [46] takes advantage of this to define a resolution-style proof procedure for SNF which is built around a temporal resolution rule for recognising looping sequences of step rules (see below) in the set of rules. Extensive research has been undertaken in implementing and improving this rather complex resolution rule, for example, Degtyarev et al. [37], Dixon [40], Gago et al. [51].

SNF has been adapted and extended to other logics, including pure future time CTL and LTL by Bolotov [14], which involves extending these logics with an additional **start** operator. This is a concept naturally expressible in logics with past. SNF has also been extended to first-order temporal logic [47] which allows SNF-based resolution to be used for infinite state verification of multi-agent systems.

However, little of this work is of direct relevance here. The correctness proofs for SNF translations given in the literature above is not sufficiently formal to allow for the extension to bounded paths we will need in the following chapter. Although transformations to SNF are described by the authors listed above, only Dixon [40] discusses them in any depth. The treatment here is somewhat more thorough.

4.1.1 Formal Definition of SNF

We introduce the operator $\Rightarrow$ as a synonym for $\rightarrow$ which allows us to distinguish syntactically between the antecedent and consequent of a rule.

Definition 4.1.1 (SNF rule)

A *rule* in SNF is an implication connecting a past time antecedent and a future time consequent. The antecedent is limited to the past time step operators **Y** and **Z** applied to a conjunction of literals while the consequent is limited to the eventuality operator **F** applied to a disjunction of literals, or a purely propositional disjunction of literals.

We define the set $\mathsf{snfrule}$ as the smallest set obtained by the production rules

$$\frac{l_0 \in \mathsf{lit}, \ldots, l_n \in \mathsf{lit}}{\mathbf{Z}\,\bot \Rightarrow \bigvee_{i=0}^{n} l_i \in \mathsf{snfrule}} \qquad \frac{l_0 \in \mathsf{lit}, \ldots, l_k \in \mathsf{lit}, \ldots, l_n \in \mathsf{lit}}{\mathbf{Y} \bigwedge_{i=0}^{k} l_i \Rightarrow \bigvee_{j=k}^{n} l_j \in \mathsf{snfrule}}$$

$$\frac{l \in \mathsf{lit}}{\mathbf{Z}\,\bot \Rightarrow \mathbf{F}\,l \in \mathsf{snfrule}} \qquad \frac{l \in \mathsf{lit}, l_0 \in \mathsf{lit}, \ldots, l_n \in \mathsf{lit}}{\mathbf{Y} \bigwedge_{i=0}^{k} l_i \Rightarrow \mathbf{F}\,l \in \mathsf{snfrule}}$$

Definition 4.1.2 (SNF formula)

A *formula* in SNF is a conjunction of rules under a **G** operator. We define the set of formulae in SNF, snf, as the smallest set obtained by the production rule

$$\frac{}{\top \in \mathsf{snf}} \qquad \frac{r_0, \ldots, r_n \in \mathsf{snfrule}}{\mathbf{G}(r_0 \wedge \cdots \wedge r_n) \in \mathsf{snf}}$$

As with CNF for propositional logic (Section 2.2.1), we allow an alternative notation for SNF as sets of rules. In this case, snf is the set of finite subsets of $\mathsf{snfrule}$, and we elide the outermost **G** operator. A set of rules Ψ corresponds to the LTL expression $\mathbf{G} \bigwedge_{\psi \in \Psi} \psi$.

SNF shares a particular characteristic with NNF: negations are applied only to atomic propositions. To reduce the complexity of transformations, we will consider only LTL formulae which are already in NNF (see Figure 2.9 for this conversion).

4.2 Recasting SNF using LTL

The use of SNF for the pure future time logic LTL was first proposed by Bolotov [14] as a stepping-stone towards $\mathrm{SNF_C}$, the projection of the normal form to CTL.

The standard definition of SNF is in terms of past-time step operators and future-time eventualities. In a pure-future logic, no past-time modalities are available. This is easily overcome by considering the particular rule types available together with the start-of-time boundary. Consider a rule $r \in \mathsf{snfrule}$, of the form $\mathbf{Y}\, p \rightarrow f$. Because of the $\mathbf{Y}$ operator, there are two cases to consider depending on the current time:

$$\models^0 \mathbf{Y}\, p \rightarrow f \Leftrightarrow \top$$
$$\models^i \mathbf{Y}\, p \rightarrow f \Leftrightarrow (\models^{i-1} p) \rightarrow (\models^i f) \qquad \text{if } i \geq 1$$

This behaviour may, however, be modelled by reinstating the $\mathbf{X}$ operator. Consider the formula $p \rightarrow \mathbf{X}\, f$. We notice that this has the same semantic interpretation of the original expression:

$$\models^{i-1} p \rightarrow \mathbf{X}\, f \Leftrightarrow (\models^{i-1} p) \rightarrow (\models^i f) \qquad \text{if } i \geq 1$$

Similarly, $\mathbf{Y}\, p \rightarrow \mathbf{F}\, f$ corresponds to $p \rightarrow \mathbf{X}\mathbf{F}\, f$. However, it is more convenient in this case to use the rule $p \rightarrow \mathbf{F}\, f$. To obtain $p \rightarrow \mathbf{X}\mathbf{F}\, f$ requires a more complex sequence of transformations. We will discuss this in more detail in Section 4.4.

The discussion above covers two of the four rule types. The remaining types of rules have $\mathbf{Z}\, \bot$ as an antecedent—an expression which is true only in state 0. Bolotov and Fisher [15] deal with this by extending LTL with an additional operator **start** designed specifically to replace this usage, giving it the semantic interpretation

$$\pi \models^i \mathbf{start} \Leftrightarrow i = 0$$

We redefine the set of rules for future-time LTL based on the above observations. We take this opportunity to divide the set, defining four distinct sets for the four types of rule that we will consider. This will simplify the encodings described in Section 5.3.1 and subsequent sections.

Definition 4.2.1 (SNF rule)
We define the sets $\mathsf{snfrule}^{\chi}_{\mathsf{L}}$ for each rule type χ as the smallest set obtained by the production rules

$$\frac{l_0 \in \mathsf{lit}, \ldots, l_n \in \mathsf{lit}}{\mathbf{start} \Rightarrow \bigvee_{i=0}^{n} l_i \in \mathsf{snfrule}^{\mathbf{start}}_{\mathsf{L}}} \qquad \frac{l_0 \in \mathsf{lit}, \ldots, l_k \in \mathsf{lit}, \ldots, l_n \in \mathsf{lit}}{\bigwedge_{i=0}^{k} l_i \Rightarrow \mathbf{X} \bigvee_{j=k}^{n} l_j \in \mathsf{snfrule}^{\mathbf{X}}_{\mathsf{L}}}$$

$$\frac{l \in \mathsf{lit}}{\mathbf{start} \Rightarrow \mathbf{F}\, l \in \mathsf{snfrule}_{\mathsf{L}}^{\mathbf{start\,F}}} \qquad \frac{l \in \mathsf{lit}, l_0 \in \mathsf{lit}, \ldots, l_n \in \mathsf{lit}}{\bigwedge_{i=0}^{k} l_i \Rightarrow \mathbf{F}\, l \in \mathsf{snfrule}_{\mathsf{L}}^{\mathbf{F}}}$$

We define the set of all rules

$$\mathsf{snfrule}_{\mathsf{L}} \doteq \mathsf{snfrule}_{\mathsf{L}}^{\mathbf{start}} \cup \mathsf{snfrule}_{\mathsf{L}}^{\mathbf{X}} \cup \mathsf{snfrule}_{\mathsf{L}}^{\mathbf{start\,F}} \cup \mathsf{snfrule}_{\mathsf{L}}^{\mathbf{F}}$$

Definition 4.2.2 (SNF formula)
A *formula* in SNF for LTL is a set of rules or a conjunction of rules under a **G** operator. We define the set of formulae in SNF, $\mathsf{snf}_{\mathsf{L}}$, as the smallest set obtained by the production rules

$$\frac{}{\top \in \mathsf{snf}_{\mathsf{L}}} \qquad \frac{r_0, \ldots, r_n \in \mathsf{snfrule}_{\mathsf{L}}}{\mathbf{G}(r_0 \wedge \cdots \wedge r_n) \in \mathsf{snf}_{\mathsf{L}}}$$

As before, in the set notation, $\mathsf{snf}_{\mathsf{L}}$ is the set of finite subsets of $\mathsf{snfrule}_{\mathsf{L}}$; a set of rules Ψ corresponds to $\mathbf{G} \bigwedge_{\psi \in \Psi} \psi$.

4.3 Denotational Semantics and QLTL

In order to properly explain and justify the transformation from LTL to SNF, it is necessary to work in terms of quantified LTL (QLTL). An SNF expression is a quantifier-free QLTL expression. We define QLTL in terms of the denotational semantics suggested by Jackson [64].

4.3.1 Quantified LTL

We extend LTL to form QLTL by adding a set of variables Q over which we will define quantification. Variables are given truth-assignments by an environment function $\rho : Q \nrightarrow 2^{\mathbb{N}}$, where $i \in \rho(q)$ if q holds in the ith state along the path. Notice that, unlike other presentations of SNF, this means that variables are independent of the model—no concept of extending the model is required. This helps to keep the presentation clean and straightforward.

The relational semantics are extended with the environment function on the left hand side:

$$\pi, \rho \models^{i} \alpha \Leftrightarrow i \in \rho(\alpha)$$

We adopt a similar notation for quantifiers to first-order logic, giving a *range* expression restricting the considered values of the quantified variable: $\exists \alpha, \phi \,.\, \psi$ indicates that only values of α for which ϕ holds are to be considered

in the search for a witness. For example, $\forall\alpha, \mathbf{F}\,\alpha \;.\; \phi$ holds if every α that eventually becomes true satisfies ϕ.

The relational semantics of the quantifiers can be given as

$$\pi, \rho \models^i \forall\alpha, \psi \;.\; \phi \Leftrightarrow \forall q \subseteq \mathbb{N},\, (\pi, \rho(\alpha \mapsto q) \models^0 \psi) \;.\; \pi, \rho(\alpha \mapsto q) \models^i \phi$$

$$\pi, \rho \models^i \exists\alpha, \psi \;.\; \phi \Leftrightarrow \exists q \subseteq \mathbb{N},\, (\pi, \rho(\alpha \mapsto q) \models^0 \psi) \;.\; \pi, \rho(\alpha \mapsto q) \models^i \phi$$

Notice that the range expressions are evaluated at time step 0 since they constrain all of the possible interpretations of the quantified variable.

The range expression $\top$ may be omitted, giving the unrestricted quantifiers $\exists\alpha \;.\; \psi$ and $\forall\alpha \;.\; \psi$. We will also allow the usual syntactic sugaring of quantifiers and their arguments, for example writing $\forall x, y \ldots$ for $\forall x \;.\; \forall y \ldots$

Definition 4.3.1 (QLTL formula)

Quantified linear temporal logic (QLTL) is an extension of LTL to include a set Q of variables and a language of quantifiers over these variables. The set $\mathsf{qltl} \supset \mathsf{ltl}$ of formulae in QLTL is the smallest set obtained by the production rules

$$\frac{}{\top, \bot \in \mathsf{qltl}} \quad \frac{a \in AP}{a \in \mathsf{qltl}} \quad \frac{\alpha \in Q}{\alpha \in \mathsf{qltl}} \quad \frac{\phi \in \mathsf{qltl}}{\neg\phi \in \mathsf{qltl}}$$

$$\frac{\phi_0, \ldots, \phi_n \in \mathsf{qltl},\; n \geq 1}{\phi_0 \wedge \cdots \wedge \phi_n \in \mathsf{qltl}} \quad \frac{\phi_0, \ldots, \phi_n \in \mathsf{qltl},\; n \geq 1}{\phi_0 \vee \cdots \vee \phi_n \in \mathsf{qltl}}$$

$$\frac{\phi_0, \phi_1 \in \mathsf{qltl}}{\phi_0 \rightarrow \phi_1 \in \mathsf{qltl}} \quad \frac{\phi_0, \phi_1 \in \mathsf{qltl}}{\phi_0 \leftrightarrow \phi_1 \in \mathsf{qltl}}$$

$$\frac{\phi \in \mathsf{qltl}}{\mathbf{X}\,\phi \in \mathsf{qltl}} \quad \frac{\phi \in \mathsf{qltl}}{\mathbf{F}\,\phi \in \mathsf{qltl}} \quad \frac{\phi \in \mathsf{qltl}}{\mathbf{G}\,\phi \in \mathsf{qltl}} \quad \frac{\phi_0, \phi_1 \in \mathsf{qltl}}{\phi_0\,\mathbf{U}\,\phi_1 \in \mathsf{qltl}} \quad \frac{\phi_0, \phi_1 \in \mathsf{qltl}}{\phi_0\,\mathbf{R}\,\phi_1 \in \mathsf{qltl}}$$

$$\frac{\phi, \psi \in \mathsf{qltl},\; \alpha \in Q}{\exists\alpha, \psi \;.\; \phi \in \mathsf{qltl}} \quad \frac{\phi, \psi \in \mathsf{qltl},\; \alpha \in Q}{\forall\alpha, \psi \;.\; \phi \in \mathsf{qltl}} \quad \frac{\phi \in \mathsf{qltl},\; \alpha \in Q}{\exists\alpha \;.\; \phi \in \mathsf{qltl}} \quad \frac{\phi \in \mathsf{qltl},\; \alpha \in Q}{\forall\alpha \;.\; \phi \in \mathsf{qltl}}$$

The set of free (unbound) variables which appear in a QLTL expression is written $fv(\phi)$. We consider free variables to be implicitly existentially quantified, and as such if an unrestricted existential quantifier appears outermost (or can be moved outermost, for example where the quantified variable does not appear in the context of the quantifier) we remove it.

4.3.2 Denotational Semantics

The meaning of a QLTL formula can be given as the set of states along the path in which is holds. This allows us to a direct mathematical meaning

to any formula in the context of a given path and a given environment. This representation means that we can understand, using set-theoretic manipulations, what it means for two formulae to be equivalent.

Definition 4.3.2 (denotational semantics of QLTL)
The denotational semantics of an QLTL formula ϕ, with respect to a model M, a path $\pi \in M$ and an environment $\rho : Q \nrightarrow 2^{\mathbb{N}}$ is written $\langle\!\langle \phi \rangle\!\rangle^{\pi,\rho} \subseteq \mathbb{N}$ and is derived from the relational semantics by

$$\langle\!\langle \phi \rangle\!\rangle^{\pi,\rho} = \{i \mid \pi, \rho \models^i \phi\}$$

and hence

$$i \in \langle\!\langle \phi \rangle\!\rangle^{\pi,\rho} \Leftrightarrow \pi, \rho \models^i \phi$$

This definition allows for some useful characterisations of LTL operators in terms of set operators, for example,

$$\begin{aligned} \langle\!\langle \phi \wedge \psi \rangle\!\rangle^{\pi,\rho} &= \langle\!\langle \phi \rangle\!\rangle^{\pi,\rho} \cap \langle\!\langle \psi \rangle\!\rangle^{\pi,\rho} \\ \langle\!\langle \phi \vee \psi \rangle\!\rangle^{\pi,\rho} &= \langle\!\langle \phi \rangle\!\rangle^{\pi,\rho} \cup \langle\!\langle \psi \rangle\!\rangle^{\pi,\rho} \\ \langle\!\langle \neg\phi \rangle\!\rangle^{\pi,\rho} &= \mathbb{N} \setminus \langle\!\langle \phi \rangle\!\rangle^{\pi,\rho} \end{aligned}$$

Although the definition of the denotational semantics is given above in terms of the relational semantics, it is also useful to see it written out in full. The identities given in Figure 4.1 follow directly from Definition 4.3.2.

We adapt equisatisfiability (Definition 2.2.8) to apply to QLTL formulae in a specialised way: we consider only variations in the environment, but assume that the path is constant for both formulae. This will turn out to be convenient: the transformations discussed later in this chapter involve the introduction of new variables but not new propositions.

Definition 4.3.3 (equisatisfiable)
QLTL formulae ϕ and ϕ' are *equisatisfiable*, written $\phi \cong \phi'$, if they are both defined on the same set of atomic propositions and if they are satisfied by the same sets of paths:

$$\phi \cong \psi \quad \doteq \quad \forall \pi \,.\, (\exists \rho \,.\, \pi, \rho \models \phi) \Leftrightarrow (\exists \rho \,.\, \pi, \rho \models \phi')$$

$$
\begin{aligned}
\langle\!\langle a \rangle\!\rangle^{\pi,\rho} &\equiv \{i \mid \pi \models^i a\} \\
\langle\!\langle \alpha \rangle\!\rangle^{\pi,\rho} &\equiv \rho(\alpha) \\
\langle\!\langle \neg\phi \rangle\!\rangle^{\pi,\rho} &\equiv \mathbb{N} \setminus \langle\!\langle \phi \rangle\!\rangle \\
\langle\!\langle \phi_0 \wedge \phi_1 \rangle\!\rangle^{\pi,\rho} &\equiv \langle\!\langle \phi_0 \rangle\!\rangle^{\pi,\rho} \cap \langle\!\langle \phi_1 \rangle\!\rangle^{\pi,\rho} \\
\langle\!\langle \phi_0 \vee \phi_1 \rangle\!\rangle^{\pi,\rho} &\equiv \langle\!\langle \phi_0 \rangle\!\rangle^{\pi,\rho} \cup \langle\!\langle \phi_1 \rangle\!\rangle^{\pi,\rho} \\
\langle\!\langle \phi_0 \rightarrow \phi_1 \rangle\!\rangle^{\pi,\rho} &\equiv (\mathbb{N} \setminus \langle\!\langle \phi_0 \rangle\!\rangle^{\pi,\rho}) \cup \langle\!\langle \phi_1 \rangle\!\rangle^{\pi,\rho} \\
\langle\!\langle \phi_0 \leftrightarrow \phi_1 \rangle\!\rangle^{\pi,\rho} &\equiv \langle\!\langle \phi_0 \rangle\!\rangle = \langle\!\langle \phi_1 \rangle\!\rangle^{\pi,\rho} \\
\langle\!\langle \mathbf{X}\,\phi \rangle\!\rangle^{\pi,\rho} &\equiv \{i \mid i+1 \in \langle\!\langle \phi \rangle\!\rangle^{\pi,\rho}\} \\
\langle\!\langle \mathbf{F}\,\phi \rangle\!\rangle^{\pi,\rho} &\equiv \{i \mid \exists j \geq i \;.\; j \in \langle\!\langle \phi \rangle\!\rangle^{\pi,\rho}\} \\
\langle\!\langle \mathbf{G}\,\phi \rangle\!\rangle^{\pi,\rho} &\equiv \{i \mid \forall j \geq i \;.\; j \in \langle\!\langle \phi \rangle\!\rangle^{\pi,\rho}\} \\
\langle\!\langle \phi_0\,\mathbf{U}\,\phi_1 \rangle\!\rangle^{\pi,\rho} &\equiv \{i \mid \exists j \geq i \;.\; j \in \langle\!\langle \phi_1 \rangle\!\rangle^{\pi,\rho} \;\wedge\; \forall n, i \leq n < j \;.\; n \in \langle\!\langle \phi_0 \rangle\!\rangle^{\pi,\rho}\} \\
\langle\!\langle \phi_0\,\mathbf{R}\,\phi_1 \rangle\!\rangle^{\pi,\rho} &\equiv \{i \mid \forall j \geq i \;.\; j \in \langle\!\langle \phi_1 \rangle\!\rangle^{\pi,\rho} \;\vee\; \exists n, i \leq n < j \;.\; n \in \langle\!\langle \phi_0 \rangle\!\rangle^{\pi,\rho}\} \\
\langle\!\langle \forall\alpha, \psi \;.\; \phi \rangle\!\rangle^{\pi,\rho} &= \bigcap_{\substack{q \subseteq \mathbb{N}, \\ 0 \in \langle\!\langle \psi \rangle\!\rangle^{\pi,\rho(\alpha \mapsto q)}}} \langle\!\langle \phi \rangle\!\rangle^{\pi,\rho(\alpha \mapsto q)} \\
\langle\!\langle \exists\alpha, \psi \;.\; \phi \rangle\!\rangle^{\pi,\rho} &= \bigcup_{\substack{q \subseteq \mathbb{N}, \\ 0 \in \langle\!\langle \psi \rangle\!\rangle^{\pi,\rho(\alpha \mapsto q)}}} \langle\!\langle \phi \rangle\!\rangle^{\pi,\rho(\alpha \mapsto q)} \\
\langle\!\langle \forall\alpha \;.\; \phi \rangle\!\rangle^{\pi,\rho} &= \bigcap_{q \subseteq \mathbb{N}} \langle\!\langle \phi \rangle\!\rangle^{\pi,\rho(\alpha \mapsto q)} \\
\langle\!\langle \exists\alpha \;.\; \phi \rangle\!\rangle^{\pi,\rho} &= \bigcup_{q \subseteq \mathbb{N}} \langle\!\langle \phi \rangle\!\rangle^{\pi,\rho(\alpha \mapsto q)}
\end{aligned}
$$

Figure 4.1: Denotational semantics of QLTL

4.3.3 Context Functions

We extend the notation for context functions defined in Section 2.5.3 to apply to QLTL in the natural way. The denotational semantics of a context function application $\langle\!\langle \Phi[\psi] \rangle\!\rangle^{\pi,\rho}$ is given by $\langle\!\langle \varphi \rangle\!\rangle^{\pi,\rho}$ where $\varphi = \Phi[\psi]$. That is, the denotational semantics of the context function application is the same as the denotational semantics of its result. We will refer to a context function itself as $\Phi[]$ although it is not a well formed QLTL expression on its own.

Considering together the denotational semantics of a context function application $\langle\!\langle \Phi[\psi] \rangle\!\rangle^{\pi,\rho}$ and of its parameter $\langle\!\langle \psi \rangle\!\rangle^{\pi,\rho}$, we can see a context function as a function $2^{\mathbb{N}} \to 2^{\mathbb{N}}$ and we can thus examine the meaning of monotonicity with respect to subset ordering. A monotonic function is one whose result always increases if its argument increases:

$$\forall \psi, \psi' \in \mathbb{N} \,.\, \langle\!\langle \psi \rangle\!\rangle^{\pi,\rho} \subseteq \langle\!\langle \psi' \rangle\!\rangle^{\pi,\rho} \to \langle\!\langle \Phi[\psi] \rangle\!\rangle^{\pi,\rho} \subseteq \langle\!\langle \Phi[\psi'] \rangle\!\rangle^{\pi,\rho}$$

Lemma 4.1 (monotonicity and polarity) *A formula $\Phi[\psi]$ is monotonic in ψ if every occurrence of subformula ψ in $\Phi[\psi]$ has positive polarity.*

Proof This follows from the definition of the denotational semantics in Figure 4.1 and the definition of polarity (Definition 2.2.11). The denotational semantics of a formula is defined in terms of membership of integers in the denotational semantics of the subformulae. Negation is encoded in the denotational semantics as $\mathbb{N} \setminus a$ for a denotation a; because $i \in (\mathbb{N} \setminus a) \Leftrightarrow i \notin a$ negation corresponds to a test of non-membership.

If ψ appears un-negated (or under an even number of negations) in $\Phi[\psi]$ then the denotational semantics of $\Phi[\psi]$ is given entirely in terms of membership $i \in \langle\!\langle \psi \rangle\!\rangle$. Thus if $\forall i \in \mathbb{N} \,.\, i \in \psi \to i \in \psi'$ then $\forall i \in \mathbb{N} \,.\, i \in \langle\!\langle \Phi[\psi] \rangle\!\rangle \to i \in \langle\!\langle \Phi[\psi'] \rangle\!\rangle$, and hence $\Phi[]$ is monotonic. □

By Definition 2.2.7, a formula in NNF has only negations applied only to atomic propositions and hence all other subformulae occur positively. This means that an NNF context function is monotonic provided the considered subformula is not an atomic proposition. For this reason, it is convenient to consider only NNF formulae for conversion to SNF. We show in Section 4.5 that the monotonicity property is preserved where required by the conversion, even though the transformations given below result in formulae that are not in NNF.

It is convenient to be able to apply context functions directly to a denotation q. This is done by introducing a new variable α not otherwise mentioned in the function, and explicitly setting its denotation in the environment. We use the shorthand $\langle\!\langle \Phi[q] \rangle\!\rangle^{\pi,\rho}$ defined below to avoid cluttering the notation.

$$\langle\!\langle \Phi[q] \rangle\!\rangle^{\pi,\rho} \quad \doteq \quad \langle\!\langle \Phi[\alpha] \rangle\!\rangle^{\pi,\rho(\alpha \mapsto q)} \quad \text{for } \alpha \text{ not used in } \Phi[]$$

The language $\mathsf{snf}_{\mathsf{L}}$ is extended to include variables and their negations in the set of literals lit.

4.3.4 Fixpoints

The definitions of least and greatest fixpoints (Section 2.5.4) can now be adapted to QLTL. We phrase the definitions in terms of the subset ordering of denotations.

We extend QLTL with the fixpoint operators, which have denotational semantics

$$\langle\!\langle \mu\alpha \,.\, \Psi[\alpha] \rangle\!\rangle^{\pi,\rho} = \bigcap \{q \subseteq \mathbb{N} \mid \langle\!\langle \Psi[q] \rangle\!\rangle^{\pi,\rho} \subseteq q\}$$
$$\langle\!\langle \nu\alpha \,.\, \Psi[\alpha] \rangle\!\rangle^{\pi,\rho} = \bigcup \{q \subseteq \mathbb{N} \mid q \subseteq \langle\!\langle \Psi[q] \rangle\!\rangle^{\pi,\rho}\}$$

4.4 Transformation to SNF

The transformation from general temporal logic to SNF is given as a series of substitution rules. We discuss the transformations given by Fisher, Bolotov, Dixon, and others informally in this chapter, justifying each by appealing to the denotational semantics of QLTL developed in Section 4.3.

While the central ideas of the transformation—unwinding the fixpoint characterisations and renaming—remain unchanged, several different transformation schemes are possible. Dixon [40] devotes an entire chapter to the topic, including a brief experimental comparison. We summarise the possibilities here and describe the transformation (not given by Dixon) which gives the best results.

In the following sections we describe the key ideas behind the transformations, allowing us to then construct the various transformations with confidence in their correctness.

4.4.1 Renaming

Renaming in a temporal context is closely related to renaming in the propositional context (see Definition 2.2.10). In the latter, we replace a subformula ϕ by a variable r_ϕ and write down the definition of r_ϕ as $r_\phi \rightarrow \phi$ (assuming ϕ has positive polarity).

In the temporal domain, the evaluation of the subformula ϕ may occur at any (or every) time step, and the definition of r_ϕ must likewise be given at every time step. We therefore write $\mathbf{G}(r_\phi \rightarrow \phi)$ for the definition. The existential quantification allows us to do this without loss of generality.

Renaming a temporal formula allows us to flatten it in order to approach the separated normal form. Consider the formula $(\mathbf{F}\, p)\, \mathbf{U} (\mathbf{G}\, q)$. Renaming each component in turn produces the definitions

$$\begin{aligned} \mathbf{G}(r_{\mathbf{F}\, p} &\rightarrow \mathbf{F}\, p) \\ \mathbf{G}(r_{\mathbf{G}\, q} &\rightarrow \mathbf{G}\, q) \\ \mathbf{G}(r_{(\mathbf{F}\, p)\, \mathbf{U} (\mathbf{G}\, q)} &\rightarrow r_{\mathbf{F}\, p}\, \mathbf{U}\, r_{\mathbf{G}\, q}) \end{aligned}$$

while the original formula has been reduced to simply $r_{(\mathbf{F}\, p)\, \mathbf{U} (\mathbf{G}\, q)}$.

4.4.1.1 Justification in Denotational Semantics

In general, we need to show that

$$\Phi[\psi] \quad \cong \quad \exists r_\psi \,.\, \Phi[r_\psi] \wedge \mathbf{G}(r_\psi \rightarrow \psi)$$

This holds provided that the following conditions hold: $\Phi[]$ is monotonic; the context in which $\Phi[\psi]$ appears is purely propositional and itself monotonic (that is, that the conditions of Lemma 4.2 below hold).

The denotation of $\mathbf{G}(r_\phi \rightarrow \phi)$ is $\{i \mid \forall j \geq i \,.\, j \in \langle\!\langle r_\phi \rangle\!\rangle^{\pi,\rho} \rightarrow j \in \langle\!\langle \phi \rangle\!\rangle^{\pi,\rho}\}$. In a non-temporal context, we consider only $0 \in \langle\!\langle \mathbf{G}(r_\phi \rightarrow \phi) \rangle\!\rangle^{\pi,\rho}$ which is hence equivalent to $\langle\!\langle r_\phi \rangle\!\rangle^{\pi,\rho} \subseteq \langle\!\langle \phi \rangle\!\rangle^{\pi,\rho}$. Following Lemma 4.1, if $\Phi[]$ is monotonic, then $0 \in \langle\!\langle \Phi[r_\psi] \rangle\!\rangle^{\pi,\rho} \wedge 0 \in \langle\!\langle \mathbf{G}(r_\psi \rightarrow \psi) \rangle\!\rangle^{\pi,\rho} \rightarrow 0 \in \langle\!\langle \Phi[\psi] \rangle\!\rangle^{\pi,\rho}$ as required.

Lemma 4.2 (renaming) *Consider a formula $F[\Phi[\psi]]$ where $F[]$ is a monotonic propositional context and $\Phi[]$ is a monotonic temporal context. ψ may be renamed with a new literal r_ψ, preserving satisfiability, giving the resulting formula $F[\Phi[r_\psi] \wedge \mathbf{G}(r_\psi \rightarrow \psi)]$.*

Proof Since the existential quantification of the new variable is outermost in the resulting formula, we elide it as for the other variables. The proof now follows as described above. □

4.4.2 Fixpoint Unwinding

The fixpoint characterisations of temporal operators are described in Section 2.5.4. For example, consider the characterisation of $\mathbf{G}\,\phi = \nu\alpha\ .\ \phi \wedge \mathbf{X}\,\alpha$. We can make a single application of $\nu\alpha\ .\ \phi \wedge \mathbf{X}\,\alpha$ to $\mathbf{G}\,\phi$ in order to obtain the identity $\mathbf{G}\,\phi = \phi \wedge \mathbf{X}\,\mathbf{G}\,\phi$. Dixon [40] advocates the use of these identities (listed below) in substitution in order to obtain the **Y** and **Z** operators required for all rule forms in the definition of SNF for PTL.

$$\mathbf{G}\,\phi \equiv \phi \wedge \mathbf{X}\,\mathbf{G}\,\phi$$
$$\mathbf{F}\,\phi \equiv \phi \vee \mathbf{X}\,\mathbf{F}\,\phi$$
$$\phi\,\mathbf{R}\,\psi \equiv \psi \vee (\phi \wedge \mathbf{X}(\phi\,\mathbf{R}\,\psi))$$
$$\phi\,\mathbf{U}\,\psi \equiv \psi \wedge (\phi \vee \mathbf{X}(\phi\,\mathbf{U}\,\psi))$$

The correctness of these equations is established directly from the fixpoint characterisations.

4.4.3 Fixpoint Characterisation

We can also use the fixpoint functions to eliminate the infinite operators altogether. For example, consider the fixpoint expression $\mathbf{G}\,\phi = \nu\alpha\ .\ \phi \wedge \mathbf{X}\,\alpha$. This suggests the substitution

$$\Phi[\mathbf{G}\,\phi] \quad \cong \quad \exists r_{\mathbf{G}\,\phi}\ .\ \Phi[r_{\mathbf{G}\,\phi}] \wedge \mathbf{G}(r_{\mathbf{G}\,\phi} \leftrightarrow \phi \wedge \mathbf{X}\,r_{\mathbf{G}\,\phi})$$

where the right hand conjunct constraining the newly introduced variable to be a fixpoint of the required expression. In fact, by exploiting certain properties of $\Phi[]$ described below, we can obtain an expression closer to the desired SNF:

$$\Phi[\mathbf{G}\,\phi] \quad \cong \quad \exists r_{\mathbf{G}\,\phi}\ .\ \Phi[r_{\mathbf{G}\,\phi}] \wedge \mathbf{G}(r_{\mathbf{G}\,\phi} \rightarrow \phi \wedge \mathbf{X}\,r_{\mathbf{G}\,\phi})$$

The existential quantification of $r_{\mathbf{G}\,\phi}$ forces the computation of the *greatest fixpoint* of $r_{\mathbf{G}\,\phi} \rightarrow \mathbf{X}(\phi \wedge r_{\mathbf{G}\,\phi})$. The *least fixpoint* computation would be handled

by universal quantification and is necessary for unfolding **F** and **U**. However, **F** may appear in the final SNF, and **U** may be reduced to **F** by the identity

$$\phi \,\mathbf{U}\, \psi = (\phi \,\mathbf{W}\, \psi) \wedge \mathbf{F}\, \psi$$

4.4.3.1 Justification in Denotational Semantics

Given the fixpoint expressions $\mu\alpha \;.\; \psi[\alpha]$ and $\nu\alpha \;.\; \psi[\alpha]$ where ψ is a context function, we derive the corresponding QLTL expressions through the denotational semantics below. As in the discussion of monotonicity, we make use of the fact that $\langle\!\langle \psi \rangle\!\rangle \subseteq \langle\!\langle \psi' \rangle\!\rangle$ may be written equivalently as $\forall i \in \mathbb{N} \;.\; i \in \langle\!\langle \psi \rangle\!\rangle \rightarrow i \in \langle\!\langle \psi' \rangle\!\rangle$ and hence $0 \in \langle\!\langle \mathbf{G}(\psi \rightarrow \psi') \rangle\!\rangle$—the evaluation of $\mathbf{G}(\psi \rightarrow \psi')$ in a purely propositional context[1]. We capture this syntactic requirement by temporarily writing $\mathbf{G}_0(\psi \rightarrow \psi')$; in practice, since we will be working in the context of languages ltlrules (see Section 4.4.4) and snf_L, this restriction on syntax is imposed by the language.

$$\begin{aligned}
\langle\!\langle \mu\alpha \;.\; \psi[\alpha] \rangle\!\rangle^{\pi,\rho} &\equiv \bigcap \{a \subseteq \mathbb{N} \mid \langle\!\langle \psi[a] \rangle\!\rangle^{\pi,\rho} \subseteq a\} \\
&\equiv \bigcap_{a \subseteq \mathbb{N}} \{i \mid \langle\!\langle \psi[a] \rangle\!\rangle^{\pi,\rho} \subseteq a \rightarrow i \in a\} \\
\text{hence, } \mu\alpha \;.\; \psi[\alpha] &\equiv \forall \alpha \;.\; \mathbf{G}_0(\psi[\alpha] \rightarrow \alpha) \rightarrow \alpha \\
\langle\!\langle \nu\alpha \;.\; \psi[\alpha] \rangle\!\rangle^{\pi,\rho} &\equiv \bigcup \{a \subseteq \mathbb{N} \mid a \subseteq \langle\!\langle \psi[a] \rangle\!\rangle^{\pi,\rho}\} \\
&\equiv \bigcup_{a \subseteq \mathbb{N}} \{i \mid a \subseteq \langle\!\langle \psi[a] \rangle\!\rangle^{\pi,\rho} \wedge i \in a\} \\
\text{hence, } \nu\alpha \;.\; \psi[\alpha] &\equiv \exists \alpha \;.\; \mathbf{G}_0(\alpha \rightarrow \psi[\alpha]) \wedge \alpha
\end{aligned}$$

We consider the final equation in the case where the fixpoint expression appears in a monotonic context function Φ. By regarding the predicate $\mathbf{G}_0(\alpha \rightarrow \psi[\alpha])$ as a constraint on the variables to be considered under the existential quantification we are able to exploit the following property

$$\langle\!\langle \Phi\Big[\bigcup_{\substack{q \subseteq \mathbb{N}, \\ 0 \in \langle\!\langle \psi[q] \rangle\!\rangle^{\pi,\rho}}} q \Big] \rangle\!\rangle^{\pi,\rho} \equiv \bigcup_{\substack{q \subseteq \mathbb{N}, \\ 0 \in \langle\!\langle \psi[q] \rangle\!\rangle^{\pi,\rho}}} \langle\!\langle \Phi[q] \rangle\!\rangle^{\pi,\rho}$$

which, by the Tarski-Knaster theorem [96], holds for monotonic Φ, to give the

[1] In a temporal context, it is possible that only a subset of time-steps be tested. Hence there is no guarantee that $\mathbf{G}(\phi \rightarrow \psi) \rightarrow \langle\!\langle \phi \rangle\!\rangle \subseteq \langle\!\langle \psi \rangle\!\rangle$ although the converse still holds.

$$\mathbf{X}\,\mathbf{F}\,\phi \equiv \mu\alpha\ .\ \mathbf{X}(\phi \vee \alpha)$$
$$\mathbf{X}\,\mathbf{G}\,\phi \equiv \nu\alpha\ .\ \mathbf{X}(\phi \wedge \alpha)$$
$$\mathbf{X}(\phi\,\mathbf{U}\,\psi) \equiv \mu\alpha\ .\ \mathbf{X}(\psi \vee (\phi \wedge \alpha))$$
$$\mathbf{X}(\phi\,\mathbf{R}\,\psi) \equiv \nu\alpha\ .\ \mathbf{X}(\psi \wedge (\phi \vee \alpha))$$
$$\mathbf{X}(\phi\,\mathbf{W}\,\psi) \equiv \nu\alpha\ .\ \mathbf{X}(\psi \vee (\phi \wedge \alpha))$$

Figure 4.2: Fixpoint characterisations of LTL operators with **X**

following derivation:

$$\begin{aligned}\Phi[\nu\alpha\ .\ \psi[\alpha]] &\equiv \Phi\left[\exists\alpha\ .\ \mathbf{G}_0(\alpha \rightarrow \psi[\alpha]) \wedge \alpha\right] \\ &\equiv \exists\alpha\ .\ \mathbf{G}_0(\alpha \rightarrow \psi[\alpha]) \wedge \Phi[\alpha]\end{aligned}$$

Note that the universal quantification in the least fixpoint case means that we will not be able to eventually remove the quantifier, and hence the SNF transformation is limited in general to the greatest fixpoint operators.

In order to reach the form required by SNF, we perform a partial unrolling of each operator before using the fixpoint characterisation. For example, $\mathbf{G}\,\psi$ is replaced by $\psi \wedge \mathbf{X}\,\mathbf{G}\,\psi$. The expression $\mathbf{X}\,\mathbf{G}\,\psi$ is rewritten most straightforwardly by observing that $\mathbf{X}\,\mathbf{G}\,\psi \equiv \mathbf{G}\,\mathbf{X}\,\psi$ and hence using the distributivity of **X** and the propositional operators,

$$\mathbf{X}(\nu\alpha\ .\ \psi \wedge \mathbf{X}\,\alpha) \equiv \nu\alpha\ .\ \mathbf{X}(\psi \wedge \alpha)$$

This type of transformation may be used for all of the LTL fixpoint characterisations (see Figure 4.2).

Lemma 4.3 (fixpoint characterisation) *Where the following substitutions are made for monotonic context* Φ *and monotonic propositional context function* F*, the satisfiability of the formula is preserved.*

$$\begin{aligned}F[\Phi[\mathbf{G}\,\psi]] &\longrightarrow F[\Phi[\psi \wedge r_{\mathbf{G}\,\psi}] \wedge \mathbf{G}(r_{\mathbf{G}\,\psi} \rightarrow \mathbf{X}(\psi \wedge r_{\mathbf{G}\,\psi}))] \\ F[\Phi[\psi_0\,\mathbf{R}\,\psi_1]] &\longrightarrow F[\Phi[\psi_1 \wedge (\psi_0 \vee r_{\psi_0\,\mathbf{R}\,\psi_1})] \\ &\qquad \wedge \mathbf{G}(r_{\psi_0\,\mathbf{R}\,\psi_1} \rightarrow \mathbf{X}(\psi_1 \wedge (\psi_0 \vee r_{\psi_0\,\mathbf{R}\,\psi_1})))] \\ F[\Phi[\psi_0\,\mathbf{U}\,\psi_1]] &\longrightarrow F[\Phi[\psi_1 \vee (\psi_0 \wedge r_{\psi_0\,\mathbf{U}\,\psi_1}) \wedge \mathbf{F}\,\psi_1] \\ &\qquad \wedge \mathbf{G}(r_{\psi_0\,\mathbf{U}\,\psi_1} \rightarrow \mathbf{X}(\psi_1 \vee (\psi_0 \wedge r_{\psi_0\,\mathbf{U}\,\psi_1})))]\end{aligned}$$

Proof By making explicit the purely propositional context, we are able to drop the syntax $\mathbf{G}_0$. As before, we elide the existential quantification. The correctness of the above substitutions now follows from the characterisations in Figure 4.2, the discussion above, and the observation that $\phi\,\mathbf{U}\,\psi = (\phi\,\mathbf{W}\,\psi) \wedge \mathbf{F}\,\psi$. □

4.4.4 Replacement of Duplicate Subformulae

Dixon [40] gives an additional transformation feature based on the identification of repeated subformulae: any transformation applied to a subformula ψ in formula ϕ can be applied to all repeated occurrences of that subformula simultaneously. This means that the introduction of extra variables can be reduced as far as possible. Dixon argues that the time spent searching for these must be carefully weighed up; we note that with appropriate implementation techniques (such as hashing) it may nevertheless be efficient. With a view to eventually encoding SNF to propositional logic, it is easy to see that it is preferable to spend the time making the identification at this stage (when the formulae are small) rather than later (when the formulae are much larger).

Since the use of context functions implicitly identifies subformulae, we obtain simultaneous replacement of duplicate subformulae without extra cost.

4.5 Conversion Strategies

We combine the techniques given above into various functions for converting general LTL to SNF. We efficiently maintain the rule structure of SNF by using a weakened form of snf_L to manipulate the formula until we arrive at a formula in snf_L. That is, we manipulate formulae of the form ltlrules defined by the production rule

$$\frac{r_1, \ldots r_n \in \mathsf{ltlrule}}{\mathbf{G}(r_1 \wedge \cdots \wedge r_n) \in \mathsf{ltlrules}}$$

or in set notation as the set of finite subsets of ltlrule, where ltlrule is the smallest set obtained by the production rule

$$\frac{\phi, \psi \in \mathsf{ltl}}{\phi \Rightarrow \psi \in \mathsf{ltlrule}}$$

Note that $\mathsf{snf}_L \subset \mathsf{ltlrules} \subset \mathsf{ltl}$; as above we use $\Rightarrow$ as a synonym for $\rightarrow$ to allow for the syntactic matching of past and future components of rules.

Given a general formula $\phi \in \mathsf{ltl}$, we obtain a member of $\mathsf{ltlrules}$ by constraining ϕ to appear at the beginning of time:

$$\{\mathbf{start} \Rightarrow \phi\} \in \mathsf{ltlrules}$$

We give transformations as functions $T_\chi : \mathsf{ltlrule} \to \mathsf{ltlrules}$, each of which transforms a given rule into a set of rules which are equisatisfiable to the original. The general form is

$$T_\chi(P \Rightarrow F) = \Psi$$

where T_χ indicates the name of the transformation being applied. A transformation procedure is given as a set of transformation functions $T^* : 2^{\mathsf{ltlrule} \to \mathsf{ltlrules}}$. We define the application of a transformation procedure to a set of rules as

$$T^*(\Psi \cup \{\phi\}) = \begin{cases} T^*\left(T_\chi(\phi) \cup \Psi\right) & \text{for some } T_\chi \in T^* \text{ such that } T(\phi) \text{ is defined} \\ \Psi & \text{otherwise} \end{cases}$$

We define the transformations such that they each apply only to a subset of the possible rules; the choice of the next rule to transform and the next transformation to apply is arbitrary and restricted only by the this applicability. The form of the transformations ensures that the transformation procedures are confluent; the proofs are given later.

Following our convention, the variables f and g appearing within the transformations indicate purely propositional (non-temporal, but not necessarily atomic) formulae. We use the variables r_ϕ for a newly introduced variable which is not used elsewhere.

4.5.1 Top-down Conversion

The top-down transformation procedure is simpler to prove (the ability to do rewrites and fixpoint characterisations depends on a much narrower variety of contexts) and implement, but as only top-level temporal operators are converted a certain amount of extra renaming is required to prepare them.

The transformations for the temporal operators follow the form of Section 4.4.3:

$$T_{\mathbf{G}\downarrow}(P \Rightarrow \mathbf{G}\,\phi) = \left\{ \begin{array}{c} P \Rightarrow \phi \wedge r_{\mathbf{X}\,\mathbf{G}\,\phi} \\ r_{\mathbf{X}\,\mathbf{G}\,\phi} \Rightarrow \mathbf{X}(\phi \wedge r_{\mathbf{X}\,\mathbf{G}\,\phi}) \end{array} \right\}$$

$$T_{\mathbf{U}\downarrow}(P \Rightarrow \phi\,\mathbf{U}\,\psi) = \left\{ \begin{array}{c} P \Rightarrow \psi \vee (\phi \wedge r_{\mathbf{X}(\phi\,\mathbf{U}\,\psi)}) \\ r_{\mathbf{X}(\phi\,\mathbf{U}\,\psi)} \Rightarrow \mathbf{X}(\psi \vee (\phi \wedge r_{\mathbf{X}(\phi\,\mathbf{U}\,\psi)})) \\ P \Rightarrow \mathbf{F}\,\psi \end{array} \right\}$$

$$T_{\mathbf{R}\downarrow}(P \Rightarrow \phi\,\mathbf{R}\,\psi) = \left\{ \begin{array}{c} P \Rightarrow \psi \wedge (\phi \vee r_{\mathbf{X}(\phi\,\mathbf{R}\,\psi)}) \\ r_{\mathbf{X}(\phi\,\mathbf{R}\,\psi)} \Rightarrow \mathbf{X}(\psi \wedge (\phi \vee r_{\mathbf{X}(\phi\,\mathbf{R}\,\psi)})) \end{array} \right\}$$

The principle challenge for the top-down transformation is obtaining rules of the appropriate form to feed into the transformations above. The following transformations follow from the standard propositional identities. Note that the transformation of disjunction only renames one disjunct; the other may be renamed due to commutativity of disjunction.

$$T_{\wedge}(P \Rightarrow \phi \wedge \psi) = \left\{ \begin{array}{c} P \Rightarrow \phi \\ P \Rightarrow \psi \end{array} \right\}$$

$$T_{\vee}(P \Rightarrow \phi \vee \psi) = \left\{ \begin{array}{c} P \Rightarrow r_\phi \vee \psi \\ r_\phi \Rightarrow \phi \end{array} \right\} \quad \text{provided } \phi \text{ is not atomic}$$

Similar transformations given below are required for the argument of $\mathbf{X}$.

$$T_{\mathbf{X}\wedge}(P \Rightarrow \mathbf{X}(\phi \wedge \psi)) = \left\{ \begin{array}{c} P \Rightarrow \mathbf{X}\,\phi \\ P \Rightarrow \mathbf{X}\,\psi \end{array} \right\}$$

$$T_{\mathbf{X}\vee}(P \Rightarrow \mathbf{X}(\phi \vee \psi)) = \left\{ \begin{array}{c} P \Rightarrow \mathbf{X}(r_\phi \vee \psi) \\ r_\phi \Rightarrow \phi \end{array} \right\} \quad \text{provided } \phi \text{ is not atomic}$$

$$T_{\mathbf{X}}(P \Rightarrow \mathbf{X}\,\phi) = \left\{ \begin{array}{c} P \Rightarrow \mathbf{X}\,r_\phi \\ r_\phi \Rightarrow \phi \end{array} \right\} \quad \text{provided } \phi \text{ is not atomic or a } \wedge \text{ or } \vee$$

Since $\mathbf{F}$ can only take a literal argument in SNF, only one case is required:

$$T_{\mathbf{F}}(P \Rightarrow \mathbf{F}\,\phi) = \left\{ \begin{array}{c} P \Rightarrow \mathbf{F}\,r_\phi \\ r_\phi \Rightarrow \phi \end{array} \right\} \quad \text{provided } \phi \text{ is not atomic}$$

The overall transformation is produced by

$$T^*_{\downarrow} = \{T_{\mathbf{G}\downarrow}, T_{\mathbf{U}\downarrow}, T_{\mathbf{R}\downarrow}, T_{\wedge}, T_{\vee}, T_{\mathbf{X}\wedge}, T_{\mathbf{X}\vee}, T_{\mathbf{X}}, T_{\mathbf{F}}\}$$

4.5.1.1 Correctness and Termination

Lemma 4.4 (confluence of $T^*_{\downarrow}$) *For all rule sets Ψ, if $\phi_1 = T^*_{\downarrow}(\Psi)$ and $\phi_2 = T^*_{\downarrow}(\Psi)$ then $\phi_1 \equiv \phi_2$.*

Proof The applicability of a transformation in $T^*_\downarrow$ is determined by the main connective on the right hand side of a rule and the side conditions. This means that for a given rule the choice of transformation is deterministic. Furthermore, the order in which given rules in the set are transformed has no effect on the result of the transformation since the transformations are defined completely in terms of the given rule. □

Lemma 4.5 (termination of $T^*_\downarrow$) *For any $\phi \in$ ltl, the procedure $T^*_\downarrow(\{\mathbf{start} \Rightarrow \text{NNF}(\phi)\})$ terminates.*

Proof Each of the transformations $T_{\mathbf{X}\wedge}$, $T_{\mathbf{X}}$, $T_{\mathbf{F}}$, and $T_\wedge$ eliminates at least one connective from the right hand side of those those rules which can be further transformed: by their definition, transformations $T_{\mathbf{X}}$ and $T_{\mathbf{F}}$ produce one rule which cannot be further transformed and one which has one fewer connective than the original rule; $T_{\mathbf{X}\wedge}$ and $T_\wedge$ similarly produce two smaller rules. Hence the number of applications of these transformations is bounded.

$T_{\mathbf{X}\vee}$ and $T_\vee$ eliminate the possibility of further application of the transformation to a given subformula; after up to two applications, the disjunction cannot be further transformed.

Each of the transformations $T_{\mathbf{G}\downarrow}$, $T_{\mathbf{U}\downarrow}$ and $T_{\mathbf{R}\downarrow}$ removes one of the temporal operators **G**, **U** or **R** from the set of rules, so each resulting rule has one fewer of these temporal operators than the original rule and hence the number of applications of these transformations is also bounded. □

Lemma 4.6 (monotonicity during $T^*_\downarrow$) *Consider the rule set Ψ obtained by transformations from $T^*_\downarrow$ applied to $\{\mathbf{start} \Rightarrow \text{NNF}(\phi)\}$ for $\phi \in$ ltl. For any rule $(P \Rightarrow \psi) \in \Psi$, and for any subformula ψ' of ψ, the context function $\Phi[\psi'] = \Psi$ is monotonic.*

Proof By Lemma 4.1, this lemma is equivalent to the claims that each transformation preserves the polarity of the right hand side of the rules produced, and that no transformation introduces negation, implication, or bi-implication on the right hand side of a rule. Each of these conditions is obvious from the definitions of the transformations. □

Lemma 4.7 (equisatisfiability of $T^*_\downarrow$) *Each transformation in $T^*_\downarrow$ applied to an applicable rule ϕ produces a set of rules Ψ which are equisatisfiable to ϕ.*

PROOF We first note that a set of rules Ψ is equivalent to the LTL formulae $\mathbf{G} \bigwedge_{\psi \in \Psi} \psi$ and hence $\bigwedge_{\psi \in \Psi} \mathbf{G}\, \psi$ by the semantics of $\mathbf{G}$.

The correctness of $T_{\mathbf{G}\downarrow}$, $T_{\mathbf{U}\downarrow}$ and $T_{\mathbf{R}\downarrow}$ now follows from Lemmas 4.6 and 4.3, and for $T_\vee$, $T_{\mathbf{X}\vee}$, $T_{\mathbf{X}}$ and $T_{\mathbf{F}}$ from Lemmas 4.6 and 4.2. $T_\wedge$ follows from the usual rules of propositional logic and $T_{\mathbf{X}\wedge}$ from their extension to LTL. □

Lemma 4.8 (resulting form of $T^*_\downarrow$) *For any $\phi \in$ ltl, $T^*_\downarrow(\{\mathbf{start} \Rightarrow \text{NNF}(\phi)\}) \in$ snf.*

PROOF It is easy to see from the definition of SNF that no formula in SNF can match any of the given transformations, and furthermore, that any formula in ltlrules \ snf can be transformed. Hence by Lemmas 4.4 and 4.5, $T^*_\downarrow$ terminates only when the set of rules is in SNF. □

Theorem 4.9 (correctness of $T^*_\downarrow$) *For any $\phi \in$ ltl, let $\phi' = T^*_\downarrow(\{\mathbf{start} \Rightarrow \text{NNF}(\phi)\})$. Then $\phi' \in$ snf and ϕ' is equisatisfiable to ϕ.*

PROOF This follows from Lemmas 4.8 and 4.7. □

4.5.1.2 Illustration

We illustrate the top-down conversion and some of its associated issues with an example. Consider the formula $\mathbf{G}\,\mathbf{F}\,a$. After putting it in the correct initial form, we transform the outermost $\mathbf{G}$ operator:

$$\left\{ \mathbf{start} \Rightarrow \mathbf{G}\,\mathbf{F}\,a \right\} \xrightarrow{T_{\mathbf{G}\downarrow}} \left\{ \begin{array}{l} \mathbf{start} \Rightarrow r_{\mathbf{X}\,\mathbf{G}\,\mathbf{F}\,a} \wedge \mathbf{F}\,a \\ r_{\mathbf{X}\,\mathbf{G}\,\mathbf{F}\,a} \Rightarrow \mathbf{X}\,(r_{\mathbf{X}\,\mathbf{G}\,\mathbf{F}\,a} \wedge \mathbf{F}\,a) \end{array} \right\}$$

then remove the resulting conjunctions (for brevity we deal with both at the same time):

$$\left\{ \begin{array}{l} \mathbf{start} \Rightarrow r_{\mathbf{X}\,\mathbf{G}\,\mathbf{F}\,a} \wedge \mathbf{F}\,a \\ \\ r_{\mathbf{X}\,\mathbf{G}\,\mathbf{F}\,a} \Rightarrow \mathbf{X}\,(r_{\mathbf{X}\,\mathbf{G}\,\mathbf{F}\,a} \wedge \mathbf{F}\,a) \\ \\ \end{array} \right\} \xrightarrow[T_{\mathbf{X}\wedge}]{T_\wedge} \left\{ \begin{array}{l} \mathbf{start} \Rightarrow r_{\mathbf{X}\,\mathbf{G}\,\mathbf{F}\,a} \\ \mathbf{start} \Rightarrow \mathbf{F}\,a \\ r_{\mathbf{X}\,\mathbf{G}\,\mathbf{F}\,a} \Rightarrow \mathbf{X}\,r_{\mathbf{X}\,\mathbf{G}\,\mathbf{F}\,a} \\ r_{\mathbf{X}\,\mathbf{G}\,\mathbf{F}\,a} \Rightarrow \mathbf{X}\,\mathbf{F}\,a \end{array} \right\}$$

and finally separate the **F** operator from its context:

$$\left\{\begin{array}{l} \mathbf{start} \Rightarrow r_{\mathbf{XGF}a} \\ \mathbf{start} \Rightarrow \mathbf{F}\,a \\ r_{\mathbf{XGF}a} \Rightarrow \mathbf{X}\,r_{\mathbf{XGF}a} \\ r_{\mathbf{XGF}a} \Rightarrow \mathbf{X}\,\mathbf{F}\,a \end{array}\right\} \xrightarrow{T_{\mathbf{X}}} \left\{\begin{array}{l} \mathbf{start} \Rightarrow r_{\mathbf{XGF}a} \\ \mathbf{start} \Rightarrow \mathbf{F}\,a \\ r_{\mathbf{XGF}a} \Rightarrow \mathbf{X}\,r_{\mathbf{XGF}a} \\ r_{\mathbf{XGF}a} \Rightarrow \mathbf{X}\,r_{\mathbf{F}a} \\ r_{\mathbf{F}a} \Rightarrow \mathbf{F}\,a \end{array}\right\}$$

A longer example, $a\,\mathbf{U}(\mathbf{G}\,b)$, is given in Figure 4.4 (page 98) without further comment.

4.5.2 Bottom-up Conversion

In the bottom-up conversion we apply transformations to the innermost temporal operators first, using constraints on the context functions to enforce the ordering. As the transformation results in the replacement of a temporal operator by a propositional subformula the technique eventually results in the transformation of all operators.

As before, the transformations for the temporal operators follow the form of Section 4.4.3; in this case, however, we enforce the "innermost" constraint by allowing only propositional arguments to the operators:

$$T_{\mathbf{G}\uparrow}(P \Rightarrow \Phi[\mathbf{G}\,f]) \doteq \left\{\begin{array}{r} P \Rightarrow \Phi\left[f \wedge r_{\mathbf{XG}f}\right] \\ r_{\mathbf{XG}f} \Rightarrow \mathbf{X}\left(f \wedge r_{\mathbf{XG}f}\right) \end{array}\right\}$$

$$T_{\mathbf{U}\uparrow}(P \Rightarrow \Phi[f\,\mathbf{U}\,g]) \doteq \left\{\begin{array}{r} P \Rightarrow \Phi\left[\left(g \vee \left(f \wedge r_{\mathbf{X}(f\,\mathbf{U}\,g)}\right)\right) \wedge \mathbf{F}\,g\right] \\ r_{\mathbf{X}(f\,\mathbf{U}\,g)} \Rightarrow \mathbf{X}(g \vee (f \wedge r_{\mathbf{X}(f\,\mathbf{U}\,g)})) \end{array}\right\}$$

$$T_{\mathbf{R}\uparrow}(P \Rightarrow \Phi[f\,\mathbf{R}\,g]) \doteq \left\{\begin{array}{r} P \Rightarrow \Phi\left[g \wedge \left(f \vee r_{\mathbf{X}(f\,\mathbf{R}\,g)}\right)\right] \\ r_{\mathbf{X}(f\,\mathbf{R}\,g)} \Rightarrow \mathbf{X}(g \wedge (f \vee r_{\mathbf{X}(f\,\mathbf{R}\,g)})) \end{array}\right\}$$

To handle **X** and **F**, we rename them whenever they appear with a propositional argument. In order to prevent the repeated application of these rules we define them in terms of a non-identity context function $\Phi^+[]$ which for all ψ has the property $\Phi^+[\psi] \neq \psi$.

$$T_{\mathbf{X}\uparrow}(P \Rightarrow \Phi^+[\mathbf{X}\,f]) \doteq \left\{\begin{array}{r} P \Rightarrow \Phi^+\left[r_{\mathbf{X}f}\right] \\ r_{\mathbf{X}f} \Rightarrow \mathbf{X}\,f \end{array}\right\}$$

$$T_{\mathbf{F}\uparrow}(P \Rightarrow \Phi^+[\mathbf{F}\, f]) \doteq \left\{ \begin{array}{l} P \Rightarrow \Phi^+\left[r_{\mathbf{F}\, f}\right] \\ r_{\mathbf{F}\, f} \Rightarrow \mathbf{F}\, f \end{array} \right\}$$

To achieve the correct propositional form of the rules, we use the transformations $T_\wedge$, $T_\vee$, $T_{\mathbf{X}\wedge}$, $T_{\mathbf{X}\vee}$ and $T_{\mathbf{F}}$ defined for the top-down conversion. The overall transformation is thus produced by

$$T^*_\uparrow = \{T_{\mathbf{G}\uparrow}, T_{\mathbf{U}\uparrow}, T_{\mathbf{R}\uparrow}, T_{\mathbf{X}\uparrow}, T_{\mathbf{F}\uparrow}, T_\wedge, T_\vee, T_{\mathbf{X}\wedge}, T_{\mathbf{X}\vee}, T_{\mathbf{F}}\}$$

4.5.2.1 Correctness and Termination

Lemma 4.10 (confluence of $T^*_\uparrow$) *For all rule sets Ψ, if $\phi_1 = T^*_\uparrow(\Psi)$ and $\phi_2 = T^*_\uparrow(\Psi)$ then $\phi_1 \equiv \phi_2$.*

Proof For the transformations which are shared with the top-down conversion ($T_\wedge$, $T_\vee$, $T_{\mathbf{X}\wedge}$, $T_{\mathbf{X}\vee}$ and $T_{\mathbf{F}}$) we appeal to Lemma 4.4.

The applicability of the remaining transformations to a given rule depend on the appearance of temporal operators with propositional arguments. There can therefore be several applicable transformations. However, the application of a transformation changes only the part of the rule which is matched by the context function: the remainder of the rule, and hence the applicability of any other transformation is unaffected. In addition, since the transformations always introduce a new variable, the application of one transformation does not change the interpretation of the context functions defined by other transformations.

For a subformula consisting of a temporal operator with a temporal argument, no transformation is applicable to the outermost temporal operator until all of the temporal operators in its argument have been transformed. By the argument above, this expansion can happen in any order with identical results. This argument easily extends to a confluence argument for general temporal formulae.

As noted in Lemma 4.4, the order in which given rules in the set are transformed has no effect on the result of the transformation. □

Lemma 4.11 (termination of $T^*_\uparrow$) *For any $\phi \in |\mathrm{tl}|$, the procedure $T^*_\uparrow(\{\mathbf{start} \Rightarrow \mathrm{NNF}(\phi)\})$ terminates.*

Proof Each of the transformations $T_{\mathbf{G}\uparrow}$, $T_{\mathbf{U}\uparrow}$ and $T_{\mathbf{R}\uparrow}$ removes one of the temporal operators **G**, **U** or **R** from the set of rules, so each resulting rule has

one fewer instance of these temporal operators than the original rule and hence the number of applications of these transformations is also bounded.

For the remaining transformations, the argument follows in the same was as for Lemma 4.5. □

Lemma 4.12 (monotonicity during $T^*_\uparrow$) *Consider the rule set Ψ obtained by transformations from $T^*_\uparrow$ applied to $\{\mathbf{start} \Rightarrow \text{NNF}(\phi)\}$ for $\phi \in \mathsf{ltl}$. For any rule $(P \Rightarrow \psi) \in \Psi$, and for any subformula ψ' of ψ, the context function $\Phi[\psi'] = \Psi$ is monotonic.*

Proof Since no transformation introduces either a negation, and implication, or a bi-implication on the right hand side of a rule, by Lemma 4.1 the monotonicity of the individual context functions defined in the transformation are preserved, and hence the monotonicity of the overall context function defined above, is preserved. □

Lemma 4.13 (equisatisfiability of $T^*_\uparrow$) *Each transformation in $T^*_\uparrow$ applied to an applicable rule ϕ produces a set of rules Ψ which are equisatisfiable to ϕ.*

Proof We first note that a set of rules Ψ is equivalent to the LTL formulae $\mathbf{G} \bigwedge_{\psi \in \Psi} \psi$ and hence $\bigwedge_{\psi \in \Psi} \mathbf{G}\, \psi$ by the semantics of $\mathbf{G}$.

The correctness of $T_{\mathbf{G}\uparrow}$, $T_{\mathbf{U}\uparrow}$ and $T_{\mathbf{R}\uparrow}$ now follows from Lemmas 4.12 and 4.3, and for $T_{\mathbf{F}\uparrow}$ and $T_{\mathbf{X}\uparrow}$ from Lemmas 4.12 and 4.2. $T_\wedge$ and $T_{\vee 1}$ follow from the usual rules of propositional logic while $T_\vee$ is a result of two applications of Lemma 4.2. □

Lemma 4.14 (resulting form of $T^*_\uparrow$) *For any $\phi \in \mathsf{ltl}$, $T^*_\uparrow(\{\mathbf{start} \Rightarrow \text{NNF}(\phi)\}) \in \mathsf{snf}$.*

Proof It is easy to see from the definition of SNF that no formula in SNF can match any of the given transformations, and furthermore, that any formula in $\mathsf{ltlrules} \setminus \mathsf{snf}$ can be transformed. Hence by Lemmas 4.10 and 4.11, $T^*_\uparrow$ terminates only when the set of rules is in SNF. □

Theorem 4.15 (correctness of $T^*_\uparrow$) *For any $\phi \in \mathsf{ltl}$, let $\phi' = T^*_\uparrow(\{\mathbf{start} \Rightarrow \text{NNF}(\phi)\})$. Then $\phi' \in \mathsf{snf}$ and ϕ' is equisatisfiable to ϕ.*

Proof This follows from Lemmas 4.14 and 4.13. □

4.5.2.2 Illustration

We illustrate the bottom-up conversion by using the same example as for the top-down conversion, $\mathbf{G}\,\mathbf{F}\,a$. After putting it in the correct initial form, we transform the innermost $\mathbf{F}$ operator:

$$\left\{\begin{array}{l}\mathbf{start} \Rightarrow \mathbf{G}\,\mathbf{F}\,a\\ \\ \end{array}\right\} \xrightarrow{T_{\mathbf{F}\uparrow}} \left\{\begin{array}{l}\mathbf{start} \Rightarrow \mathbf{G}\,r_{\mathbf{F}\,a}\\ r_{\mathbf{F}\,a} \Rightarrow \mathbf{F}\,a\end{array}\right\}$$

then the remaining $\mathbf{G}$ operator:

$$\left\{\begin{array}{l}\mathbf{start} \Rightarrow \mathbf{G}\,r_{\mathbf{F}\,a}\\ \\ r_{\mathbf{F}\,a} \Rightarrow \mathbf{F}\,a\end{array}\right\} \xrightarrow[T_{\mathbf{X}\wedge}]{T_{\wedge}} \left\{\begin{array}{l}\mathbf{start} \Rightarrow r_{\mathbf{X}\,\mathbf{G}\,r_{\mathbf{F}\,a}} \wedge r_{\mathbf{F}\,a}\\ r_{\mathbf{X}\,\mathbf{G}\,r_{\mathbf{F}\,a}} \Rightarrow \mathbf{X}(r_{\mathbf{X}\,\mathbf{G}\,r_{\mathbf{F}\,a}} \wedge r_{\mathbf{F}\,a})\\ r_{\mathbf{F}\,a} \Rightarrow \mathbf{F}\,a\end{array}\right\}$$

and finally expand the conjunctions:

$$\left\{\begin{array}{l}\mathbf{start} \Rightarrow r_{\mathbf{X}\,\mathbf{G}\,r_{\mathbf{F}\,a}} \wedge r_{\mathbf{F}\,a}\\ \\ r_{\mathbf{X}\,\mathbf{G}\,r_{\mathbf{F}\,a}} \Rightarrow \mathbf{X}(r_{\mathbf{X}\,\mathbf{G}\,r_{\mathbf{F}\,a}} \wedge r_{\mathbf{F}\,a})\\ \\ r_{\mathbf{F}\,a} \Rightarrow \mathbf{F}\,a\end{array}\right\} \xrightarrow{T_{\mathbf{X}\wedge}} \left\{\begin{array}{l}\mathbf{start} \Rightarrow r_{\mathbf{X}\,\mathbf{G}\,r_{\mathbf{F}\,a}}\\ \mathbf{start} \Rightarrow r_{\mathbf{F}\,a}\\ r_{\mathbf{X}\,\mathbf{G}\,r_{\mathbf{F}\,a}} \Rightarrow \mathbf{X}(r_{\mathbf{X}\,\mathbf{G}\,r_{\mathbf{F}\,a}})\\ r_{\mathbf{X}\,\mathbf{G}\,r_{\mathbf{F}\,a}} \Rightarrow \mathbf{X}(r_{\mathbf{F}\,a})\\ r_{\mathbf{F}\,a} \Rightarrow \mathbf{F}\,a\end{array}\right\}$$

The second example is given in Figure 4.5 (page 100; to improve the layout, we write $r_{\mathbf{X}\,\mathbf{U}}$ for $r_{\mathbf{X}(a\,\mathbf{U}(b\wedge r_{\mathbf{X}\,\mathbf{G}\,b}))}$.

4.5.3 Discussion: the Propositional Form

The key difference between the results of the two transformations given is not in the number of rules produced (since both produce the same number of rules). It is more interesting to consider the reason for introducing new variables and hence new rules, either for the fixpoint characterisation, or for renaming of subformulae. In the top-down case, in the large example the last fixpoint characterisation is made as the second to last transformation, producing six rules. In the bottom-up case it is the second transformation, producing four rules. While the propositional manipulation is necessary in the top-down case in order to be able to apply the fixpoint transformations, in the bottom-up case they are required only to obtain the precise propositional form. In addition,

the transformations in question ($T_\wedge$, $T_\vee$, $T_{\mathbf{X}\wedge}$, $T_{\mathbf{X}\vee}$ and $T_{\mathbf{F}}$) are top-down in nature, and it is thus compelling to drop them. The resulting transformation procedure,

$$T^*_\uparrow = \{T_{\mathbf{G}\uparrow}, T_{\mathbf{U}\uparrow}, T_{\mathbf{R}\uparrow}, T_{\mathbf{X}\uparrow}, T_{\mathbf{F}\uparrow}\}$$

is similar to $T^*_\uparrow$ but results in an equisatisfiable formula which is not in true SNF. The resulting form, which we will call PSNF (for *propositional SNF*) is similar to SNF without the propositional restrictions: temporal operators apply to general NNF propositional formulae, and general NNF propositional formulae may also appear on the left hand side of rules. We also allow purely propositional rules of the form $f \Rightarrow g$ for $f, g \in \mathsf{nnf}$ in PSNF.

Theorem 4.16 (correctness of $T^*_\uparrow$) *For any $\phi \in |\mathrm{tl}|$, let $\phi' = T^*_\uparrow(\{\mathbf{start} \Rightarrow \textsc{Nnf}(\phi)\})$. Then ϕ' is equisatisfiable to ϕ.*

Proof This follows from Lemmas 4.14 and 4.13. □

4.5.3.1 Combined Transformation

While a top-down procedure is easy to implement—working as it does on the main connective of each rule—the bottom-up procedure introduces fewer additional atomic propositions to the formula. Unfortunately, finding the candidate subformula to process at each iteration of $T^*_\uparrow$ takes linear time in general. We show here how a top-down procedure can be used to implement the bottom-up transformation without introducing any extra algorithmic complexity.

We generalise the transformation by use of a pair of functions $P : |\mathrm{tl}| \to \mathsf{prop}$ and $D : |\mathrm{tl}| \to |\mathrm{tl}|\mathsf{rules}$ given in Figure 4.3. The former, rewriting an LTL subformula as a propositional one, allows us to write transformations with an assurance that the temporal operators will be removed; the latter produces the definitions of any new variables introduced by the former function.

The function D relies for its definition on P as introduced propositions are constrained by formulae which themselves require transformation. To preserve the algorithmic complexity of the transformation, we therefore require a dynamic programming or memoisation approach to the implementation of P. For the implementation used to generate the results in Chapter 7, we use a pointer-based approach which allows us to replace all occurrences of ϕ with $P(\phi)$ simultaneously.

$$
\begin{aligned}
P(\mathbf{G}\,\phi) &= P(\phi) \wedge r_{\mathbf{X}\,\mathbf{G}\,\phi} \\
P(\phi\,\mathbf{R}\,\psi) &= P(\psi) \wedge (P(\phi) \vee r_{\mathbf{X}(\phi\,\mathbf{R}\,\psi)}) \\
P(\phi\,\mathbf{U}\,\psi) &= (P(\psi) \vee (P(\phi) \wedge r_{\mathbf{X}(\phi\,\mathbf{U}\,\psi)})) \wedge P(\mathbf{F}\,\psi) \\
P(\mathbf{F}\,\phi) &= r_{\mathbf{F}\,\phi} \\
P(\mathbf{X}\,\phi) &= r_{\mathbf{X}\,\phi} \\
P(\phi \wedge \psi) &= P(\phi) \wedge P(\psi) \\
P(\phi \vee \psi) &= P(\phi) \vee P(\psi) \\
P(\neg a) &= \neg a \\
P(a) &= a
\end{aligned}
$$

$$
\begin{aligned}
D(\mathbf{G}\,\phi) &= \{r_{\mathbf{X}\,\mathbf{G}\,\phi} \Rightarrow \mathbf{X}(P(\phi) \wedge r_{\mathbf{X}\,\mathbf{G}\,\phi})\} \cup D(\phi) \\
D(\phi\,\mathbf{R}\,\psi) &= \{r_{\mathbf{X}(\phi\,\mathbf{R}\,\psi)} \Rightarrow \mathbf{X}(P(\psi) \wedge (P(\phi) \vee r_{\mathbf{X}(\phi\,\mathbf{R}\,\psi)}))\} \cup D(\phi) \cup D(\psi) \\
D(\phi\,\mathbf{U}\,\psi) &= \{r_{\mathbf{X}(\phi\,\mathbf{U}\,\psi)} \Rightarrow \mathbf{X}(P(\psi) \vee (P(\phi) \wedge r_{\mathbf{X}(\phi\,\mathbf{U}\,\psi)}))\} \cup D(\phi) \cup D(\psi) \\
D(\mathbf{F}\,\phi) &= \{r_{\mathbf{F}\,\phi} \Rightarrow \mathbf{F}\,P(\phi)\} \cup D(\phi) \\
D(\mathbf{X}\,\phi) &= \{r_{\mathbf{X}\,\phi} \Rightarrow \mathbf{X}\,P(\phi)\} \cup D(\phi) \\
D(\phi \wedge \psi) &= D(\phi) \cup D(\psi) \\
D(\phi \vee \psi) &= D(\phi) \cup D(\psi) \\
D(\neg a) &= \emptyset \\
D(a) &= \emptyset
\end{aligned}
$$

Figure 4.3: Top-down implementation of PSNF conversion

Given an LTL formula ϕ, we may now obtain the equivalent PSNF expression as

$$\{\mathbf{start} \Rightarrow P(\phi)\} \cup D(\phi)$$

Theorem 4.17 (correcness of *P* and *D*) *The set* $\{\mathbf{start} \Rightarrow P(\text{NNF}(\phi))\} \cup D(\phi)$ *is equivalent to that obtained by the conversion* $T^*_{\uparrow}(\{\mathbf{start} \Rightarrow \text{NNF}(\phi)\})$.

Proof For general $\phi \in |\text{tl}|$ we show that the theorem holds by induction on the structure of ϕ. The base cases are straightforward:

$$T_{\mathbf{G}\uparrow}(\{P \Rightarrow \Phi[\mathbf{G}\, f]\}) = \{P \Rightarrow \Phi[P(\mathbf{G}\, f)]\} \cup D(\mathbf{G}\, f)$$
$$T_{\mathbf{R}\uparrow}(\{P \Rightarrow \Phi[f\, \mathbf{R}\, g]\}) = \{P \Rightarrow \Phi[P(f\, \mathbf{R}\, g)]\} \cup D(f\, \mathbf{R}\, f)$$
$$T_{\mathbf{U}\uparrow}(\{P \Rightarrow \Phi[f\, \mathbf{U}\, g]\}) = \{P \Rightarrow \Phi[P(f\, \mathbf{U}\, g)]\} \cup D(f\, \mathbf{U}\, g)$$
$$T_{\mathbf{X}\uparrow}(\{P \Rightarrow \Phi[\mathbf{X}\, f]\}) = \{P \Rightarrow \Phi[P(\mathbf{X}\, f)]\} \cup D(\mathbf{X}\, f)$$
$$T_{\mathbf{F}\uparrow}(\{P \Rightarrow \Phi[\mathbf{F}\, f]\}) = \{P \Rightarrow \Phi[P(\mathbf{F}\, f)]\} \cup D(\mathbf{F}\, f)$$

For the step cases, we make use of 'unfolding' identities for P and D. For example, for $\mathbf{G}$ we see that

$$P(\mathbf{G}\, \phi) \equiv P(\mathbf{G}\, P(\phi)) \quad \text{and} \quad D(\mathbf{G}\, \phi) \equiv D(\mathbf{G}\, P(\phi)) \cup D(\phi)$$

and hence

$$T_{\mathbf{G}\uparrow}(\{P \Rightarrow \Phi[\mathbf{G}\, \phi]\}) \equiv T_{\mathbf{G}}(\{P \Rightarrow \Phi[\mathbf{G}\, P(\phi)]\}) \cup D(\phi)$$

We can make similar observations for the other temporal operators.

Since $T^*_{\uparrow}(D(\phi)) \equiv D(\phi)$, we deduce that $T^*_{\uparrow}(\{\mathbf{start} \Rightarrow \phi\}) = \{\mathbf{start} \Rightarrow P(\phi)\} \cup D(\phi)$ □

4.6 Summary

In this chapter we presented SNF, a clausal normal form for temporal logic. The recasting of SNF for LTL is based on work by Bolotov [14], but the justification for the transformation to SNF is new work.

The conversion to SNF, like some conversions from propositional logic to CNF, involves the introduction of new symbols. We add quantifiers to LTL to form QLTL and give its denotational semantics. The expressiveness of QLTL is required to derive and explain the transformations, but during the

transformation all quantifiers are existentials and are moved to the outermost position in the formula. Such quantifiers are removed and left implicit, so the result of conversion to SNF is a formula in quantifier-free QLTL. This approach is in contrast to the presentations of Fisher [46] and Bolotov [14], where a single set of variables and propositions is used.

The transformation to SNF is given in terms of two operations: renaming and fixpoint characterisation. We prove that these result in equisatisfiable formulae using the denotational semantics. Transformations functions are given in terms of rewrites. Two basic transformations are given: the top-down transformation $T^*_{\downarrow}$ which expands the outermost connective first, and the bottom-up transformation $T^*_{\uparrow}$ which expands the innermost connective first. The former is simpler to describe and fast to exectute, but the latter results in the introduction of fewer variables.

The general form of SNF includes restrictions on the form of propositional subformulae. The transformations to achieve these are reminiscent of the conversion to clause form, so we argue that a less restrictive normal form, PSNF, is more appropriate for eventual use in BMC. PSNF does not restrict the propositional subformulae beyond NNF. With this in mind, we give a top-down procedure to achieve the bottom-up transformation to PSNF, thus obtaining the best of both worlds.

The figures on the following pages are examples of the top-down and bottom-up conversion processes.

$$\left\{\mathbf{start} \Rightarrow a\,\mathbf{U}(\mathbf{G}\,b)\right\}$$

$$\xrightarrow{T_{\mathbf{U}\downarrow}}$$

$$\left\{\begin{array}{l}\mathbf{start} \Rightarrow \mathbf{G}\,b \vee (a \wedge r_{a\,\mathbf{U}(\mathbf{G}\,b)}) \\ r_{a\,\mathbf{U}(\mathbf{G}\,b)} \Rightarrow \mathbf{X}(\mathbf{G}\,b \vee (a \wedge r_{a\,\mathbf{U}(\mathbf{G}\,b)})) \\ \mathbf{start} \Rightarrow \mathbf{F}\,\mathbf{G}\,b\end{array}\right\}$$

$$\xrightarrow{T_{\vee},\,T_{\mathbf{X}\vee}}$$

$$\left\{\begin{array}{l}\mathbf{start} \Rightarrow r_{\mathbf{G}\,b} \vee (a \wedge r_{a\,\mathbf{U}(\mathbf{G}\,b)}) \\ r_{\mathbf{G}\,b} \Rightarrow \mathbf{G}\,b \\ r_{a\,\mathbf{U}(\mathbf{G}\,b)} \Rightarrow \mathbf{X}(r_{\mathbf{G}\,b} \vee (a \wedge r_{a\,\mathbf{U}(\mathbf{G}\,b)})) \\ \mathbf{start} \Rightarrow \mathbf{F}\,\mathbf{G}\,b\end{array}\right\}$$

$$\xrightarrow{T_{\vee},\,T_{\mathbf{X}\vee}}$$

$$\left\{\begin{array}{l}\mathbf{start} \Rightarrow r_{\mathbf{G}\,b} \vee r_{a\wedge r_{a\,\mathbf{U}(\mathbf{G}\,b)}} \\ r_{a\wedge r_{a\,\mathbf{U}(\mathbf{G}\,b)}} \Rightarrow a \wedge r_{a\,\mathbf{U}(\mathbf{G}\,b)} \\ r_{\mathbf{G}\,b} \Rightarrow \mathbf{G}\,b \\ r_{a\,\mathbf{U}(\mathbf{G}\,b)} \Rightarrow \mathbf{X}(r_{\mathbf{G}\,b} \vee r_{a\wedge r_{a\,\mathbf{U}(\mathbf{G}\,b)}}) \\ \mathbf{start} \Rightarrow \mathbf{F}\,\mathbf{G}\,b\end{array}\right\}$$

Figure 4.4: Example conversion of $a\,\mathbf{U}(\mathbf{G}\,b)$ to SNF

$$\xrightarrow{T_{\mathbf{F}}}$$

$$\left\{\begin{array}{rcl}
\textbf{start} & \Rightarrow & r_{\mathbf{G}\,b} \vee r_{a \wedge r_{a\,\mathbf{U}(\mathbf{G}\,b)}} \\
r_{a \wedge r_{a\,\mathbf{U}(\mathbf{G}\,b)}} & \Rightarrow & a \wedge r_{a\,\mathbf{U}(\mathbf{G}\,b)} \\
r_{\mathbf{G}\,b} & \Rightarrow & \mathbf{G}\,b \\
r_{a\,\mathbf{U}(\mathbf{G}\,b)} & \Rightarrow & \mathbf{X}(r_{\mathbf{G}\,b} \vee r_{a \wedge r_{a\,\mathbf{U}(\mathbf{G}\,b)}}) \\
\textbf{start} & \Rightarrow & \mathbf{F}\,r_{\mathbf{G}\,b}
\end{array}\right\}$$

$$\xrightarrow{T_{\mathbf{G}\downarrow}}$$

$$\left\{\begin{array}{rcl}
\textbf{start} & \Rightarrow & r_{\mathbf{G}\,b} \vee r_{a \wedge r_{a\,\mathbf{U}(\mathbf{G}\,b)}} \\
r_{a \wedge r_{a\,\mathbf{U}(\mathbf{G}\,b)}} & \Rightarrow & a \wedge r_{a\,\mathbf{U}(\mathbf{G}\,b)} \\
r_{\mathbf{G}\,b} & \Rightarrow & b \wedge r_{\mathbf{X}\,\mathbf{G}\,b} \\
r_{\mathbf{X}\,\mathbf{G}\,b} & \Rightarrow & \mathbf{X}(b \wedge r_{\mathbf{X}\,\mathbf{G}\,b}) \\
r_{a\,\mathbf{U}(\mathbf{G}\,b)} & \Rightarrow & \mathbf{X}(r_{\mathbf{G}\,b} \vee r_{a \wedge r_{a\,\mathbf{U}(\mathbf{G}\,b)}}) \\
\textbf{start} & \Rightarrow & \mathbf{F}\,r_{\mathbf{G}\,b}
\end{array}\right\}$$

$$\xrightarrow{T_{\wedge},\, T_{\mathbf{X}\wedge}}$$

$$\left\{\begin{array}{rcl}
\textbf{start} & \Rightarrow & r_{\mathbf{G}\,b} \vee r_{a \wedge r_{a\,\mathbf{U}(\mathbf{G}\,b)}} \\
r_{a \wedge r_{a\,\mathbf{U}(\mathbf{G}\,b)}} & \Rightarrow & a \\
r_{a \wedge r_{a\,\mathbf{U}(\mathbf{G}\,b)}} & \Rightarrow & r_{a\,\mathbf{U}(\mathbf{G}\,b)} \\
r_{\mathbf{G}\,b} & \Rightarrow & b \\
r_{\mathbf{G}\,b} & \Rightarrow & r_{\mathbf{X}\,\mathbf{G}\,b} \\
r_{\mathbf{X}\,\mathbf{G}\,b} & \Rightarrow & \mathbf{X}(b) \\
r_{\mathbf{X}\,\mathbf{G}\,b} & \Rightarrow & \mathbf{X}(r_{\mathbf{X}\,\mathbf{G}\,b}) \\
r_{a\,\mathbf{U}(\mathbf{G}\,b)} & \Rightarrow & \mathbf{X}(r_{\mathbf{G}\,b} \vee r_{a \wedge r_{a\,\mathbf{U}(\mathbf{G}\,b)}}) \\
\textbf{start} & \Rightarrow & \mathbf{F}\,r_{\mathbf{G}\,b}
\end{array}\right\}$$

Figure 4.4 (continued)

$$\left\{ \mathbf{start} \Rightarrow a\,\mathbf{U}(\mathbf{G}\,b) \right\}$$

$$\xrightarrow{T_{\mathbf{G}\uparrow}}$$

$$\left\{ \begin{array}{l} \mathbf{start} \Rightarrow a\,\mathbf{U}(b \wedge r_{\mathbf{X}\mathbf{G}\,b}) \\ r_{\mathbf{X}\mathbf{G}\,b} \Rightarrow \mathbf{X}(b \wedge r_{\mathbf{X}\mathbf{G}\,b}) \end{array} \right\}$$

$$\xrightarrow{T_{\mathbf{U}\uparrow}}$$

$$\left\{ \begin{array}{l} \mathbf{start} \Rightarrow (b \wedge r_{\mathbf{X}\mathbf{G}\,b}) \vee (a \wedge r_{\mathbf{X}\mathbf{U}}) \\ \quad r_{\mathbf{X}\mathbf{U}} \Rightarrow \mathbf{X}((b \wedge r_{\mathbf{X}\mathbf{G}\,b}) \vee (a \wedge r_{\mathbf{X}\mathbf{U}})) \\ \mathbf{start} \Rightarrow \mathbf{F}(b \wedge r_{\mathbf{X}\mathbf{G}\,b}) \\ r_{\mathbf{X}\mathbf{G}\,b} \Rightarrow \mathbf{X}(b \wedge r_{\mathbf{X}\mathbf{G}\,b}) \end{array} \right\}$$

$$\xrightarrow{T_{\vee}, T_{\mathbf{X}\vee}}$$

$$\left\{ \begin{array}{l} \quad \mathbf{start} \Rightarrow r_{b \wedge r_{\mathbf{X}\mathbf{G}\,b}} \vee (a \wedge r_{\mathbf{X}\mathbf{U}}) \\ r_{b \wedge r_{\mathbf{X}\mathbf{G}\,b}} \Rightarrow b \wedge r_{\mathbf{X}\mathbf{G}\,b} \\ \quad r_{\mathbf{X}\mathbf{U}} \Rightarrow \mathbf{X}(r_{b \wedge r_{\mathbf{X}\mathbf{G}\,b}} \vee (a \wedge r_{\mathbf{X}\mathbf{U}})) \\ \quad \mathbf{start} \Rightarrow \mathbf{F}(b \wedge r_{\mathbf{X}\mathbf{G}\,b}) \\ \quad r_{\mathbf{X}\mathbf{G}\,b} \Rightarrow \mathbf{X}(b \wedge r_{\mathbf{X}\mathbf{G}\,b}) \end{array} \right\}$$

Figure 4.5: Example conversion of $a\,\mathbf{U}(\mathbf{G}\,b)$ to SNF

$$\xrightarrow{T_{\vee},\, T_{\mathbf{X}\vee}}$$

$$\left\{\begin{array}{rl}
\textbf{start} & \Rightarrow r_{b\wedge r_{\mathbf{XG}b}} \vee r_{a\wedge r_{\mathbf{XU}}} \\
r_{b\wedge r_{\mathbf{XG}b}} & \Rightarrow b \wedge r_{\mathbf{XG}b} \\
r_{a\wedge r_{\mathbf{XU}}} & \Rightarrow a \wedge r_{\mathbf{XU}} \\
r_{\mathbf{XU}} & \Rightarrow \mathbf{X}(r_{b\wedge r_{\mathbf{XG}b}} \vee r_{a\wedge r_{\mathbf{XU}}}) \\
\textbf{start} & \Rightarrow \mathbf{F}(b \wedge r_{\mathbf{XG}b}) \\
r_{\mathbf{XG}b} & \Rightarrow \mathbf{X}(b \wedge r_{\mathbf{XG}b})
\end{array}\right\}$$

$$\xrightarrow{T_{\mathbf{F}}}$$

$$\left\{\begin{array}{rl}
\textbf{start} & \Rightarrow r_{b\wedge r_{\mathbf{XG}b}} \vee r_{a\wedge r_{\mathbf{XU}}} \\
r_{b\wedge r_{\mathbf{XG}b}} & \Rightarrow b \wedge r_{\mathbf{XG}b} \\
r_{a\wedge r_{\mathbf{XU}}} & \Rightarrow a \wedge r_{\mathbf{XU}} \\
r_{\mathbf{XU}} & \Rightarrow \mathbf{X}(r_{b\wedge r_{\mathbf{XG}b}} \vee r_{a\wedge r_{\mathbf{XU}}}) \\
\textbf{start} & \Rightarrow \mathbf{F}\, r_{b\wedge r_{\mathbf{XG}b}} \\
r_{\mathbf{XG}b} & \Rightarrow \mathbf{X}(b \wedge r_{\mathbf{XG}b})
\end{array}\right\}$$

$$\xrightarrow{T_{\wedge},\, T_{\mathbf{X}\wedge}}$$

$$\left\{\begin{array}{rl}
\textbf{start} & \Rightarrow r_{b\wedge r_{\mathbf{XG}b}} \vee r_{a\wedge r_{\mathbf{XU}}} \\
r_{b\wedge r_{\mathbf{XG}b}} & \Rightarrow b \\
r_{b\wedge r_{\mathbf{XG}b}} & \Rightarrow r_{\mathbf{XG}b} \\
r_{a\wedge r_{\mathbf{XU}}} & \Rightarrow a \\
r_{a\wedge r_{\mathbf{XU}}} & \Rightarrow r_{\mathbf{XU}} \\
r_{\mathbf{XU}} & \Rightarrow \mathbf{X}(r_{b\wedge r_{\mathbf{XG}b}} \vee r_{a\wedge r_{\mathbf{XU}}}) \\
\textbf{start} & \Rightarrow \mathbf{F}\, r_{b\wedge r_{\mathbf{XG}b}} \\
r_{\mathbf{XG}b} & \Rightarrow \mathbf{X}\, b \\
r_{\mathbf{XG}b} & \Rightarrow \mathbf{X}\, r_{\mathbf{XG}b}
\end{array}\right\}$$

Figure 4.5 (continued)

Chapter 5

Using the Separated Normal Form for BMC

SNF, as described in Chapter 4, is a clausal representation of temporal logic. By converting a formula to SNF, the variety of temporal operators is restricted (to **X** and **F**) as is the scope of their arguments (to propositional logic). We discuss the adaptation to SNF necessary for it to be used as part of the BMC encoding process in Section 5.2, using SNF as a preprocessing step to simplify the specification before encoding. With the SNF transformation in this position, however, the conversion to propositional logic may be simplified considerably: much of the complexity of the BMC encoding in Chapter 3 is due to the variety of temporal operators and their arguments. We examine this in more detail in Section 5.3.1.

In this chapter we focus on the partial transformation $T^*_{\restriction}$ (see Section 4.5.3). The form, PSNF, resulting from this transformation requires further changes only at the propositional level in order to reach SNF, but is much more compact that SNF itself. The route we take in this chapter is to define an encoding on the partial SNF form; the remaining transformations then overlap with the conversion from propositional logic to CNF which is discussed much more thoroughly in Chapter 6.

5.1 Motivation

Designing an encoding for bounded model checking may be seen as the task of finding an efficient way of converting an expression from a specification in a

temporal logic to a set of propositional clauses that implement that specification with respect to a propositional representation of time. The encoding presented in Chapter 3 achieves this by constructing a formula in propositional logic which is equivalent to the specification interpreted over a particular path. We have seen, however, how temporal logic also has a clause form, SNF (see Chapter 4), for which the conversion is well understood. An alternative encoding, therefore, could be to convert to SNF and then project the temporal clauses onto a given path to achieve the corresponding set of propositional clauses.

This has a series of conceptual advantages. Firstly, as we saw in Section 2.2.1, it is difficult to find the ideal conversion from general propositional logic to clause form; using SNF eliminates the large propositional representation of the specification and lifts much of the complexity of the clause form conversion to the temporal logic level, where the formula is significantly simpler. As noted in Chapter 3, a naïve propositional representation of a temporal formula with n symbols has size $O(n^k)$; this can clearly be overcome to a certain extent by the methods discussed, but the SNF method sidesteps the problem entirely. An SNF based encoding has the advantage of breaking the encoding process into smaller, more easily understood steps: the transition from the temporal domain to the propositional takes place over a much more restricted set of operators. Compared to the encoding function for BMC given in Tables 3.1 and 3.2 this conversion is very simple.

5.2 SNF Formulae over Prefix and Loop Paths

In Section 3.1 we noted that BMC is a model checking procedure for general LTL over a restricted subset of paths: those expressible using a bounded number of states. The alternative view is to give a modified semantics for LTL which take into account the particular type of path under consideration. As formulae in SNF are well-formed quantifier-free QLTL formulae, we can easily adapt the modified semantics by simply incorporating the environment ρ. This has a number of drawbacks. Most importantly, as SNF has the general form $\mathbf{G}(\ldots)$, the interpretation in the sound prefix semantics (see Section 3.2) is simply $\bot$. That is, using the sound prefix semantics of LTL to interpret SNF formulae results in all formulae being interpreted as false. While this is indeed

a sound interpretation, it is an under-approximation which is not particularly useful.

We therefore re-examine the k-prefix semantics for LTL, taking into consideration the restricted form of SNF formulae.

5.2.1 Denotational Semantics of Quantifier-free QLTL

In Section 4.3 we introduced the denotational semantics of QLTL, and this was later extended with quantified variables which were given denotations by the environment function $\rho : Q \to 2^{\mathbb{N}}$. The restricted semantics discussed in Chapter 3 are easily extended to quantifier-free QLTL by incorporating the environment function ρ into the definition of each semantic judgement.

Although the range of ρ is infinite subsets of $\mathbb{N}$, the k-prefix semantics are defined on members of the sets less than or equal to k; we write[1] $\varrho = \rho|_k$ for the finite restriction of ρ, and for symmetry with ϖ, we will write $|\varrho| = k$ for $ran(\varrho) = \{0 \ldots k\}$. For k-l-loop paths, we enforce the loop property (similar to Definition 3.1.4):

$$\mathrm{Loop}_{k,l}(\rho) = \forall a \in dom(\rho), \forall i > k \;.\; i \in \rho(a) \Leftrightarrow i - k + l \in \rho(a)$$

so that the denotation can be represented with a finite number of propositional variables.

5.2.2 k-Prefix Paths

In Section 3.1.1.1 we presented sound k-prefix semantics of LTL which are *maximally complete*: any semantics which is strictly more complete than that given is not sound for general LTL expressions. However, the semantics given are not sufficiently expressive to be useful for encoding PSNF, so we develop a suitable semantics which is sound only for the restricted syntax of PSNF.

With the exception of the **G** operator, the relational semantics given in Figure 5.1 follows naturally from the semantics of QLTL. The construction is the same as for LTL in Section 3.1.1.1. The **G** operator only occurs as the outermost connective in PSNF expressions; we show below that in this context, interpreting it as a quantification over the first $k + 1$ states is sound.

[1] The symbol ϱ, LaTeX's "variant ρ" is used to distinguish finite environments from infinite ones. See also page xvi.

$$
\begin{array}{lcl}
\varpi, \varrho \models_k^i a & \Leftrightarrow & a \in \varpi(i) \\
\varpi, \varrho \models_k^i \neg a & \Leftrightarrow & a \notin \varpi(i) \\
\varpi, \varrho \models_k^i \alpha & \Leftrightarrow & i \in \varrho(\alpha) \\
\varpi, \varrho \models_k^i \neg\alpha & \Leftrightarrow & i \notin \varrho(\alpha) \\
\varpi, \varrho \models_k^i \phi \wedge \psi & \Leftrightarrow & (\varpi, \varrho \models_k^i \phi) \wedge (\varpi, \varrho \models_k^i \psi) \\
\varpi, \varrho \models_k^i \phi \vee \psi & \Leftrightarrow & (\varpi, \varrho \models_k^i \phi) \vee (\varpi, \varrho \models_k^i \psi) \\
\varpi, \varrho \models_k^i \mathbf{start} & \Leftrightarrow & (i = 0) \\
\varpi, \varrho \models_k^i \mathbf{X}\,\phi & \Leftrightarrow & \begin{cases} \varpi, \varrho \models_k^{i+1} \phi & \text{if } i < k \\ \bot & \text{otherwise} \end{cases} \\
\varpi, \varrho \models_k^i \mathbf{F}\,\phi & \Leftrightarrow & \exists j, i \leq j \leq k \;.\; \varpi, \varrho \models_k^j \phi \\
\varpi, \varrho \models_k^i \mathbf{G}\,\phi & \Leftrightarrow & \forall j, i \leq j \leq k \;.\; \varpi, \varrho \models_k^j \phi
\end{array}
$$

Figure 5.1: The sound semantics of quantifier-free QLTL for k-prefix paths, specialised to PSNF. The model M is implicit, with $\varpi \in M$ and $|\varpi| = k + 1$; the environment ϱ has the property $|\varrho| = k + 1$

Firstly, we show that PSNF formulae produced by the transformations have a particular form: the antecedent is always a positive (non-negated) single variable α or the operator **start**. This restriction is sufficient to allow us to show the soundness of the semantics given.

Lemma 5.1 *Given an LTL formula ϕ, each rule in the corresponding PSNF formula $\phi' \in T^*_{\uparrow}(\{\mathbf{start} \Rightarrow \text{NNF}(\phi)\})$ is of one of the following forms:*

$$\alpha \Rightarrow \mathbf{X}\, f \qquad \mathbf{start} \Rightarrow \mathbf{X}\, f \qquad \alpha \Rightarrow f$$
$$\alpha \Rightarrow \mathbf{F}\, f \qquad \mathbf{start} \Rightarrow \mathbf{F}\, f$$

where $\alpha \in fv(\phi')$.

PROOF We prove this by induction on $T^*_{\uparrow}(\phi)$: examining the constituent transformations in Section 4.5.2 we see that the only resulting rules have either the same antecedent as the original rule, or a new variable or **start** as an antecedent. Since the initial rule has **start** as the antecedent, we deduce that the lemma holds. □

Note that it is not generally the case that SNF formulae have this form. For example, SNF permits $a \wedge b \Rightarrow \mathbf{X}\, c$ even though, as shown above, expressions of this form do not occur as a result of the transformations given in the previous chapter[2].

The soundness lemma follows from this observation about the form. Notice that we consider all free variables in a QLTL formula to be implicitly existentially quantified. This is the reason for the unusual form of the soundness statement in the Lemma: we require extensions of satisfying bounded paths to be satisfy the infinite semantics, but we only consider the *existence* of an extension to the environment.

Lemma 5.2 (soundness of the PSNF bounded semantics) *For a formula ϕ of the form given in Lemma 5.1, the semantics given in Figure 5.1 is sound. That is,*

$$\forall \phi, \forall M, \forall \varrho, |\varrho| = k+1 \;.\; \forall \varpi \in M, |\varpi| = k+1 \;.$$
$$(\varpi, \varrho \models_k \phi) \rightarrow (\forall \pi \in M, \pi|_k = \varpi \;.\; \exists \rho, \rho|_k = \varrho \;.\; \pi, \rho \models \phi)$$

[2]This type of expression forms part of SNF because it is required for representing PLTL formulae (see Section 2.5.1 and Cimatti, Roveri, and Sheridan [24]). They are also formed during the decision procedures defined by Fisher [46] and others.

Proof Formula ϕ is of the form $\mathbf{G} \bigwedge_j \psi_j$, where the possible rules ψ_j are given in Lemma 5.1. Hence,

$$(\varpi, \varrho \models_k \phi) \Leftrightarrow \forall i, i \leq k \;.\; \bigwedge_j (\varpi, \varrho \models_k^i \psi_j)$$

However, notice that each ψ_j has a variable α or **start** on the left hand side. Since every variable has $i \notin \varrho(\alpha)$ for $i > k$ by definition, every rule ψ_j in states $i > k$ has a left hand side equivalent to $\bot$.

Taking $\rho = \varrho$ as the witness to the existential on the right hand side, we see that

$$(\varpi, \rho \models \phi) \Leftrightarrow \forall i \;.\; \bigwedge_j (\pi, \varrho \models^i \psi_j) \Leftrightarrow \forall i, i \leq k \;.\; \bigwedge_j (\pi, \varrho \models^i \psi_j)$$

We now show, by induction, that for any rule ψ_j at time $i \leq k$, any extension of a k-bounded path that satisfies its k-bounded semantics satisfies its unbounded semantics. The base cases (variables and propositions, and their negations) are trivial; the two cases of interest here are $\mathbf{X}$ and $\mathbf{F}$, since they are the only relevant differences between the bounded and unbounded semantics. Since $\mathbf{X}$ always occurs with positive polarity, a rule containing $\mathbf{X}$ is satisfied only if a successor state exists (such a rule cannot be satisfied at time k), and hence is satisfied in the unbounded semantics too. If an $\mathbf{F}$ is satisfied in the bounded semantics then its argument occurs between times i and k, which would be sufficient to satisfy it in the unbounded semantics.

If every rule is satisfied in the bounded semantics in the first k states, then by the above reasoning, every rule is satisfied in the unbounded semantics in the first k states. By the nature of the chosen environment, this means that every rule is satisfied in every state in the unbounded semantics. □

5.2.3 k-Loop Paths

As noted in Section 3.1.2, the semantics of LTL may be directly interpreted over k-loop paths, as they are a subset of the infinite paths. This clearly also holds for SNF, and so we can adapt the projected semantics given in Figure 3.5 in a straightforward way (Figure 5.2).

$$
\begin{array}{lcl}
\varpi, \varrho \models^{\circ i}_{k,l} a & \Leftrightarrow & a \in \varpi(i) \\
\varpi, \varrho \models^{\circ i}_{k,l} \alpha & \Leftrightarrow & i \in \varrho(\alpha) \\
\varpi, \varrho \models^{\circ i}_{k,l} \neg\phi & \Leftrightarrow & \varpi, \varrho \not\models^{i}_{k,l} \phi \\
\varpi, \varrho \models^{\circ i}_{k,l} \phi \wedge \psi & \Leftrightarrow & (\varpi, \varrho \models^{\circ i}_{k,l} \phi) \wedge (\varpi, \varrho \models^{\circ i}_{k,l} \psi) \\
\varpi, \varrho \models^{\circ i}_{k,l} \phi \vee \psi & \Leftrightarrow & (\varpi, \varrho \models^{\circ i}_{k,l} \phi) \vee (\varpi, \varrho \models^{\circ i}_{k,l} \psi) \\
\varpi, \varrho \models^{i}_{k} \mathbf{start} & \Leftrightarrow & (i = 0) \\
\varpi, \varrho \models^{\circ i}_{k,l} \mathbf{X}\,\phi & \Leftrightarrow & \varpi, \varrho \models^{\circ \rho_0(i+1)}_{k,l} \phi \\
\varpi, \varrho \models^{\circ i}_{k,l} \mathbf{F}\,\phi & \Leftrightarrow & \exists j, \min(i,l) \leq j < k \;.\; \varpi, \varrho \models^{\circ j}_{k,l} \phi \\
\varpi, \varrho \models^{\circ i}_{k,l} \mathbf{G}\,\phi & \Leftrightarrow & \forall j, \min(i,l) \leq j < k \;.\; \varpi, \varrho \models^{\circ j}_{k,l} \phi
\end{array}
$$

Figure 5.2: The projected semantics of quantifier-free QLTL for k-loop paths. The model M is implicit, with $\varpi \in M$ and $|\varpi| = k + 1$; the environment ϱ has the property $|\varrho| = k + 1$ and $\mathrm{Loop}_{k,l}(\varrho)$

5.3 SNF and BMC

Having established the use of SNF in the context of the types of path used in BMC, we now turn to the use of SNF as an encoding method for BMC. There are several issues that must be addressed. Firstly, throughout Chapter 4 we considered the conversion of LTL expressions to SNF in isolation; we must now consider the conversion in the context of the BMC encoding. Secondly, in Chapter 3 we define the BMC conversion for LTL expressions, but the result of the transformation to SNF is a quantifier-free expression in QLTL. In particular, the set of variables is given values separately from the set of atomic propositions. We therefore require a set of propositions in the encoding which is additional to those used in representing the path.

The incorporation of the environment function ϱ into the BMC formula from Definition 3.1.6 is straightforward.

Definition 5.3.1 (PSNF bounded model checking problem)
The PSNF bounded model checking problem with bound k for a model M and an PSNF formula ϕ is to determine whether a finite prefix path of

length k or a k-loop path exists in M which satisfies ϕ:

$$\mathrm{BMC}'(M,\phi,k) \doteq \exists \varpi \in M, |\varpi| = k+1 \;.\; \exists \varrho, |\varrho| = k+1 \;.\; \mathrm{BMC}'(\phi,k,\varpi,\varrho)$$

$$\mathrm{BMC}'(\phi,k,\varpi,\varrho) \doteq \varpi,\varrho \models_k \phi \vee (\exists l < k \;.\; \varpi(k) = \varpi(l) \wedge \mathrm{Loop}_{k,l}(\varrho) \wedge \varpi,\varrho \mathrel{\dot{\models}}_{k,l} \phi)$$

Lemma 5.3 (SNF transformation in a BMC context) *For all models M, all LTL formulae ϕ, and all bounds k,*

$$\mathrm{BMC}(M,\phi,k) \cong \mathrm{BMC}'(M, T^*_{\restriction}(\{\mathbf{start} \Rightarrow \mathrm{NNF}(\phi)\}), k)$$

Proof Lemmas 4.2 and 4.3 state that the SNF transformations hold provided the context in which they occur is purely propositional. We thus deduce that Theorem 4.16 continues to hold in the context of the BMC encoding. □

5.3.1 Encoding PSNF for BMC

We analyse the PSNF BMC problem BMC′ with respect to the set Ψ, where $\Psi = T^*_{\restriction}(\{\mathbf{start} \Rightarrow \mathrm{NNF}(\phi)\})$. Writing the outermost $\mathbf{G}$ explicitly, we obtain:

$$\mathrm{BMC}'(M,\Psi,k) = \exists \varpi \in M, |\varpi| = k+1 \;.\; \exists \varrho, |\varrho| = k+1 \;.\; \mathrm{BMC}'(\Psi,k,\varpi,\varrho)$$

$$\mathrm{BMC}'(\Psi,k,\varpi,\varrho) = \left(\varpi,\varrho \models_k \mathbf{G} \bigwedge_{\psi\in\Psi} \psi\right) \vee \left(\exists l < k \;.\; \varpi(k) = \varpi(l) \wedge \mathrm{Loop}_{k,l}(\varrho) \wedge \varpi,\varrho \mathrel{\dot{\models}}_{k,l} \mathbf{G} \bigwedge_{\psi\in\Psi} \psi\right)$$

We derive a propositional expression as in Section 3.2 by rewriting quantifiers with propositional connectives:

$$\mathrm{BMC_p}'(\hat{M},\Psi,k) \doteq [\![\hat{M}]\!]_k \wedge \Big({}_-[\![\mathbf{G} \bigwedge_{\psi\in\Psi} \psi]\!]^0_k \vee \bigvee_{l<k} \Big({}_l[\![\varpi(k) = \varpi(l)]\!]^0_k \wedge {}_l[\![\mathrm{Loop}_{k,l}(\varrho)]\!]_k \wedge {}_l[\![\mathbf{G} \bigwedge_{\psi\in\Psi} \psi]\!]^0_k \Big) \Big)$$

and analyse the parts which remain to be converted to propositional logic.

Table 5.1: The BMC encoding for PSNF rules, prefix path case

ϕ	$[\![\phi]\!]_k^i$
a	a^i
α	α^i
$\neg\phi$	$\neg[\![\phi]\!]_k^i$
$\phi \wedge \psi$	$[\![\phi]\!]_k^i \wedge [\![\psi]\!]_k^i$
$\phi \vee \psi$	$[\![\phi]\!]_k^i \vee [\![\psi]\!]_k^i$
$\mathbf{start} \Rightarrow f$	$i > 0 \vee [\![f]\!]_k^0$
$\mathbf{start} \Rightarrow \mathbf{X}\, f$	$i > 0 \vee [\![f]\!]_k^1$
$\mathbf{start} \Rightarrow \mathbf{F}\, f$	$i > 0 \vee \bigvee_{j=0}^{k} [\![f]\!]_k^j$
$f \Rightarrow g$	$[\![f]\!]_k^i \rightarrow [\![g]\!]_k^i$
$f \Rightarrow \mathbf{X}\, g$	$[\![f]\!]_k^i \rightarrow (i < k \wedge [\![g]\!]_k^{i+1})$
$f \Rightarrow \mathbf{F}\, g$	$[\![f]\!]_k^i \rightarrow \bigvee_{j=i}^{k} [\![g]\!]_k^j$

5.3.1.1 Propositions

In Section 3.2.1, we observed that the set of atomic propositions $\bigcup_{0\leq i\leq k} A^i$ was necessary for the encoding of a BMC problem involving the symbolic Kripke structure $\hat{M} = \langle A, \hat{I}, \hat{T}\rangle$. To encode quantifier-free QLTL we additionally need a set of propositions to capture the variables in Q. Adopting the same naming convention, we define atomic propositions $\{\alpha^i \mid \alpha \in Q\} = Q^i$ and so use the set of propositions

$$\bigcup_{0\leq i\leq k} A^i \cup \bigcup_{0\leq i\leq k} Q^i$$

in the encoding.

5.3.1.2 k-Prefix Paths

Consider the first disjunct from $\mathrm{BMC_p}'(\hat{M}, \phi, k)$

$$_{-}[\![\phi]\!]_k^0$$

The semantics in Figure 5.1 relate ϕ over a k-prefix path to a QTPL expression. As the quantifiers used are over linear restrictions of natural number variables, we replace quantifiers with conjunctions or disjunctions, the variables and linear restrictions moving to the metalanguage. Each proposition $a \in \varpi(i)$ is encoded as the proposition $a^i \in A^i$, and the environment mapping for each variable $\alpha \in fv(\phi)$, written $i \in \varrho(\alpha)$, as the proposition α^i.

The encoding is given by converting the outermost **G** according to Figure 5.1,

$$[\![\mathbf{G} \bigwedge_{\psi \in \Psi} \psi]\!]_k \quad \Leftrightarrow \quad \bigwedge_{i=0}^{k} \bigwedge_{\psi \in \Psi} [\![\psi]\!]_k^i$$

where the component rules are encoded according to Table 3.1 giving the per-rule encoding shown in Table 5.1.

5.3.1.3 k-Loop Paths

The second disjunction includes the loopback conditions on the path and the environment ${}_l[\![\varpi(k) = \varpi(l)]\!]_k^0 \wedge {}_l[\![\mathrm{Loop}_{k,l}(\varrho)]\!]_k^0$ and the encoding of the QLTL property over k-loop paths:

$${}_l[\![\phi]\!]_k^0$$

The loopback condition for the path is given in Section 3.2.1. For the environment, we require that that every variable has the same interpretation at time l and k:

$${}_l[\![\mathrm{Loop}_{k,l}(\varrho)]\!]_k^0 \quad \Leftrightarrow \quad \bigwedge_{\alpha \in Q} \alpha^l \leftrightarrow \alpha^k$$

As before, we notice that the semantics given in Figure 5.2 can used to generate a propositional encoding by replacing quantifiers with conjunctions and disjunctions and reinterpreting the linear restrictions of variables as part of the metalanguage. We use the encoding for the outermost **G** from Table 3.2 to produce

$${}_l[\![\mathbf{G} \bigwedge_{\psi \in \Psi} \psi]\!]_k = \bigwedge_{i=0}^{k} \bigwedge_{\psi \in \Psi} {}_l[\![\psi]\!]_k^i$$

with the per-rule encoding shown in Table 5.2.

In the k-loop path case, this encoding has worst-case size which is cubic with respect to k: for a rule of the form $f \Rightarrow \mathbf{F}\, g$ we consider k loopback points, then encode the **F** operator ($O(k)$ connectives) at each of k time steps. All other rules encode to quadratic size as each instance produces $O(1)$ connectives. For

Table 5.2: The BMC encoding for PSNF rules, loop path case

ϕ	${}_l[\![\phi]\!]_k^i$
a	a^i
α	α^i
$\neg\phi$	$\neg\, {}_l[\![\phi]\!]_i^k$
$\phi \wedge \psi$	${}_l[\![\phi]\!]_k^i \wedge {}_l[\![\psi]\!]_k^i$
$\phi \vee \psi$	${}_l[\![\phi]\!]_k^i \vee {}_l[\![\psi]\!]_k^i$
$\mathbf{start} \Rightarrow f$	$i > 0 \vee {}_l[\![f]\!]_k^0$
$\mathbf{start} \Rightarrow \mathbf{X}\, f$	$i > 0 \vee {}_l[\![f]\!]_k^1$
$\mathbf{start} \Rightarrow \mathbf{F}\, f$	$i > 0 \vee \bigvee_{j=0}^{k} {}_l[\![f]\!]_k^j$
$f \Rightarrow g$	${}_l[\![f]\!]_k^i \rightarrow {}_l[\![f]\!]_k^i$
$f \Rightarrow \mathbf{X}\, g$	${}_l[\![f]\!]_k^i \rightarrow \left(i < k \wedge {}_l[\![f]\!]_k^{i+1}\right) \vee \left(i = k \wedge {}_l[\![f]\!]_k^l\right)$
$f \Rightarrow \mathbf{F}\, g$	${}_l[\![f]\!]_k^i \rightarrow \bigvee_{j=\min(i,l)}^{k} {}_l[\![f]\!]_k^j$

the finite prefix path case the situation is similar: the worst case is $O(k^2)$ (rules of the form $f \Rightarrow \mathbf{F}\, g$), while the other rules are $O(k)$.

5.3.1.4 Correctness

We show that the BMC encoding for PSNF is a correct propositional representation of the PSNF BMC formula given in Definition 5.3.1. The relationship between BMC using PSNF and infinite state model checking with PSNF then follows from the soundness results given earlier in this chapter; the relationship with general infinite state model checking follows from the correctness of the PSNF transformation in the previous chapter.

The theorem and proof below follow the form of Theorem 3.2.

Theorem 5.4 (correctness of BMC encoding) *$\sigma \models \mathrm{BMC_p}'(\hat{M}, \phi, k)$ if and only if σ represents a path $\varpi \in K(\hat{M})$ and environment ϱ which satisfies $\mathrm{BMC}'(\phi, k, \varpi, \varrho)$.*

Proof The set of propositions in $\mathrm{BMC_p}(\hat{M}, \phi, k)$ is $\bigcup_{0 \le i \le k} A^i \cup \bigcup_{0 \le i \le k} Q^i$ where A is the set of atomic propositions in the symbolic Kripke structure $\hat{M}$ and Q is the set of free variables in the QLTL formulae ϕ. Writing $\lhd$ for the domain restriction operator, we can write the path represented by σ as $\varpi = A^0 \lhd \sigma, A^1 \lhd \sigma, \ldots A^k \lhd \sigma$ and the environment represented by σ as $i \in \varrho(q) \Leftrightarrow \langle q^i, \top \rangle \in \sigma$ for all $q \in Q$.

The proof now breaks into four parts corresponding to the four parts of $\mathrm{BMC_p}'(\hat{M}, \phi, k)$: we show that the solutions σ corresponds to the valid bounded paths in $K(\hat{M})$; that the solutions which satisfy the prefix encoding correspond to the paths and environments which satisfy the prefix semantics; that the solutions which satisfy the loop properties correspond to the k-loop paths and environments; and that the solutions which satisfy the k-loop encoding correspond to the paths and environments which satisfy the k-loop semantics.

1. By appealing to the definition of $\mathrm{BMC_p}'(\hat{M}, \phi, k)$ and the definition of $K(\hat{M})$ in Section 2.4.2, we see that $\mathrm{BMC_p}'(\hat{M}, \phi, k)$ constrains A^0 as strongly as $\varpi \in K(\hat{M})$ constrains $\varpi(0)$; similarly, $\mathrm{BMC_p}'(\hat{M}, \phi, k)$ constrains A^i and A^{i+1} as strongly as $\varpi \in K(\hat{M})$ constrains $\varpi(i)$ and $\varpi(i+1)$.

2. The encoding given in Table 5.1 represents precisely the same constraint on propositions $\bigcup_{0\leq i\leq k} A^i$ as the semantics in Figure 5.1 does on the path $\varpi = A^0 \lhd \sigma, A^1 \lhd \sigma, \ldots A^k \lhd \sigma$, and on propositions $\bigcup_{0\leq i\leq k} Q^i$ as the environment given by $i \in \varrho(q) \Leftrightarrow \langle q^i, \top\rangle \in \sigma$. This can be shown trivially by induction, with base cases of the atomic propositions and their negations; since the range of every quantifier is finite they are equivalent to the conjunctions and disjunctions in the encoding.

3. The constraint equating the propositions in A^l and A^k is equivalent to constraining ϖ to being a k-loop path by the observation at the start of Section 3.1.3. Similarly, the constraint equating the propositions in Q^l and Q^k is equivalent to constraining ϱ to being an environment such that $\mathrm{Loop}_{k,l}(\varrho)$ as observed in Section 5.3.1.3.

4. The encoding given in Table 5.2 represents precisely the same constraint on propositions $\bigcup_{0\leq i\leq k} A^i$ as the semantics in Figure 5.2 does on the path $\varpi = A^0 \lhd \sigma, A^1 \lhd \sigma, \ldots A^k \lhd \sigma$, and on propositions $\bigcup_{0\leq i\leq k} Q^i$ as the environment given by $i \in \varrho(q) \Leftrightarrow \langle q^i, \top\rangle \in \sigma$. As before, this can be shown trivially by induction. □

5.3.2 Improved Encodings

Although this direct encoding given above is an improvement over the monolithic encoding given in Section 3.2, it is much larger than the simple improvements we described subsequently. To improve the encoding further we notice that the encodings of Figures 5.1 and 5.2 are simple in all but a few cases, and hence we consider the common factors of the two encodings.

The first step is to abstract the encoding as

$$[\![\hat{M}]\!]_k \wedge \mathrm{enc}_c(\phi, k) \wedge \left(\mathrm{enc}_n(\phi, k) \vee \bigvee_{l<k} \left({}_l[\![\varpi(k) = \varpi(l)]\!]_k^0 \wedge {}_l[\![\mathrm{Loop}_{k,l}(\varrho)]\!]_k \wedge \mathrm{enc}_l(\phi, k, l)\right)\right)$$

where $\mathrm{enc}_c(\phi, k)$ represents the propositional encoding of ϕ which is common to both the finite prefix path case and the k-loop path case; $\mathrm{enc}_n(\phi, k)$ and $\mathrm{enc}_l(\phi, k, l)$ represent the additional constraints necessary to achieve the propositional encodings of f for the finite prefix path case and the k-loop path case

respectively. In general,

$$\mathrm{enc}_n(\phi, k) \wedge \mathrm{enc}_c(\phi, k) = [\![\phi]\!]_k$$
$$\mathrm{enc}_l(\phi, k, l) \wedge \mathrm{enc}_c(\phi, k) = {}_l[\![\phi]\!]_k$$

We annotate the encoding functions with superscript integers to distinguish the encodings presented.

5.3.2.1 Basic Factorisation

We notice that the differences between the encodings described above for the different path types are minimal, at particular time steps.

Lemma 5.5 (propositional factorisation) *For $f \in \mathsf{prop}$, $\forall l \in \mathbb{N} \;.\; {}_l[\![f]\!]^i_k \equiv [\![f]\!]^i_k$. That is, the encoding of propositional logic is the same for finite prefix and k-loop paths.*

Proof This is obvious by inspection of Figures 5.1 and 5.2. □

Lemma 5.6 (factorisation of non–F-rules) *For rules $\psi \in \mathsf{snfrule_L} \setminus \mathsf{snfrule^F_L}$, if $i < k$ then $\forall l \in \mathbb{N} \;.\; {}_l[\![\psi]\!]^i_k \equiv [\![\psi]\!]^i_k$. That is, rules other than eventuality rules encode the same for finite prefix and k-loop paths except at time k.*

Proof This is obvious by inspection of Figures 5.1 and 5.2 and by appeal to the usual propositional identities. □

We can thus make the encoding depend on the rule type:

$$\mathrm{enc}^1_c(\phi, k) = \bigwedge_{i=0}^{k-1} \bigwedge_{\psi \in \Psi \setminus \mathsf{snfrule^F_L}} [\![\psi]\!]^i_k$$
$$\mathrm{enc}^1_n(\phi, k) = \left(\bigwedge_{\psi \in \Psi \setminus \mathsf{snfrule^F_L}} [\![\psi]\!]^k_k \right) \wedge \left(\bigwedge_{i=0}^{k} \bigwedge_{\psi \in \Psi \cap \mathsf{snfrule^F_L}} [\![\psi]\!]^i_k \right)$$
$$\mathrm{enc}^1_l(\phi, k, l) = \left(\bigwedge_{\psi \in \Psi \setminus \mathsf{snfrule^F_L}} {}_l[\![\psi]\!]^k_k \right) \wedge \left(\bigwedge_{i=0}^{k} \bigwedge_{\psi \in \Psi \cap \mathsf{snfrule^F_L}} {}_l[\![\psi]\!]^i_k \right)$$

where $\Psi = T^*_{\restriction}(\{\mathbf{start} \Rightarrow \mathrm{NNF}(\phi)\})$.

This encoding has not improved the asymptotic complexity of the encoding, but sets the scene for the further improvements described below.

5.3.2.2 Improved Treatment of Eventualities

As observed for the standard BMC encoding in Section 3.4, although there are a quadratic number of instantiations of each eventuality rule, there are clearly only a linear number of *different* instantiations. This observation can be used to reduce the asymptotic size complexity from cubic to quadratic in k.

Exploiting the identity implied by Lemma 5.5, eventuality rules are encoded in the finite prefix case and the k-loop case as

$$[\![f]\!]^i_k \rightarrow \bigvee_{j=i}^{k} [\![g]\!]^j_k \qquad \text{and} \qquad [\![f]\!]^i_k \rightarrow \bigvee_{j=\min(i,l)}^{k} [\![g]\!]^j_k$$

respectively—notice that the dependency on l has been confined to the ranging of j. We identify the expression $\bigvee_j^k [\![g]\!]^j_k$ as the common factor and introduce a new set of atomic propositions $r^{*i}_{\mathbf{F}\,g}$ with the defining formulae

$$r^{*i}_{\mathbf{F}\,g} \rightarrow \bigvee_{j=i}^{k} [\![g]\!]^j_k$$

We can adapt the encoding to accommodate this:

$$\mathrm{enc}^2_c(\phi, k) = \left(\bigwedge_{i=0}^{k-1} \bigwedge_{\psi \in \Psi \setminus \mathsf{snfrule}^{\mathbf{F}}_{\mathsf{L}}} [\![\psi]\!]^i_k \right) \wedge \left(\bigwedge_{i=0}^{k} \bigwedge_{(f \Rightarrow \mathbf{F}\, g) \in \Psi} r^{*i}_{\mathbf{F}\,g} \rightarrow \bigvee_{j=i}^{k} [\![g]\!]^j_k \right)$$

$$\mathrm{enc}^2_n(\phi, k) = \left(\bigwedge_{\psi \in \Psi \setminus \mathsf{snfrule}^{\mathbf{F}}_{\mathsf{L}}} [\![\psi]\!]^k_k \right) \wedge \left(\bigwedge_{i=0}^{k} \bigwedge_{(f \Rightarrow \mathbf{F}\, g) \in \Psi} [\![f]\!]^i_k \rightarrow r^{*i}_{\mathbf{F}\,g} \right)$$

$$\mathrm{enc}^2_l(\phi, k, l) = \left(\bigwedge_{\psi \in \Psi \setminus \mathsf{snfrule}^{\mathbf{F}}_{\mathsf{L}}} {}_l[\![\psi]\!]^k_k \right) \wedge \left(\bigwedge_{i=0}^{k} \bigwedge_{(f \Rightarrow \mathbf{F}\, g) \in \Psi} [\![f]\!]^i_k \rightarrow r^{*\min(i,l)}_{\mathbf{F}\,g} \right)$$

where $\Psi = T^*_{\restriction}(\{\mathbf{start} \Rightarrow \mathrm{NNF}(\phi)\})$.

5.3.3 A Linear-Space Encoding

The encoding given above remains quadratic because rules containing $\mathbf{F}$ cannot be factored out. In order to remove the final quadratic factor, we make the following important observation.

Lemma 5.7 *If π is a k-l-loop, then $\pi \models^i \mathbf{F}\, g$ if and only if $(\pi \models^i \mathbf{F}\, g) \vee (\pi \models^l \mathbf{F}\, g)$*

Proof This follows from the semantics of eventualities and the nature of k-loop paths. The "only if" case is trivial from the usual semantics of propositional logic. We consider the "if" case.

Consider $i < l$. Then $\pi \models^l \mathbf{F}\, g$ implies $\pi \models^i \mathbf{F}\, g$ since there exists a $j \geq l$ such that $\pi \models^j g$.

Consider $l \leq i < k$. From the nature of the k-loop path, $\pi \models^l \mathbf{F}\, g$ if and only if $\pi \models^k \mathbf{F}\, g$. The latter implies $\pi \models^i \mathbf{F}\, g$ by the same argument above.

All other cases, $i \geq k$ are covered by the nature of the k-loop path. □

The crux of this lemma is that the explicit $\min(i, l)$ operation—which makes the rules in $\mathsf{snfrule}_{\mathsf{L}}^{\mathbf{F}}$ depend on l, and hence leads to the quadratic blowup—can be avoided by using a disjunction of evaluations at i and at l. Using the observation made in the previous section, we can write

$$ {}_l[\![\mathbf{F}\, f]\!]_k^i \doteq [\![\mathbf{F}\, f]\!]_k^i \vee [\![\mathbf{F}\, f]\!]_k^l $$

and we can write a similar formula for the k-prefix path,

$$ [\![\mathbf{F}\, f]\!]_k^i \doteq [\![\mathbf{F}\, f]\!]_k^i \vee \bot $$

Notice that the left hand disjuncts are the same for both encodings, but the right hand disjuncts differ. We can rename the right hand disjuncts by introducing a new atomic proposition $r_{\prec \mathbf{F}\, f}$, parameterised by the propositional argument of the eventuality, f:

$$ [\![\mathbf{F}\, f]\!]_k^i = {}_l[\![\mathbf{F}\, f]\!]_k^i = [\![\mathbf{F}\, f]\!]_k^i \vee r_{\prec \mathbf{F}\, f} $$

with the defining clauses

$$ r_{\prec \mathbf{F}\, f} \rightarrow [\![\mathbf{F}\, f]\!]_k^l \qquad \text{and} \qquad r_{\prec \mathbf{F}\, f} \rightarrow \bot $$

in the loop and prefix cases respectively. Notice that $r_{\prec \mathbf{F}\, f}$ is a single atomic proposition; copies are not formed for each state as for the propositions representing the path.

This leads to the encoding given below. Notice that in the conjunctions we match rules in $\Psi \cap \mathsf{snfrule}_{\mathsf{L}}^{\mathbf{F}}$ against the expression $(f \Rightarrow \mathbf{F}\, g)$ to extract the components of the rule.

$$ \mathrm{enc}_c^3(\phi, k) = \bigwedge_{i=0}^{k-1} \bigwedge_{\psi \in \Psi \setminus \mathsf{snfrule}_{\mathsf{L}}^{\mathbf{F}}} [\![\psi]\!]_k^i \wedge \left(\bigwedge_{i=0}^{k} \bigwedge_{(f \Rightarrow \mathbf{F}\, g) \in \Psi} [\![f]\!]_k^i \rightarrow [\![\mathbf{F}\, g]\!]_k^i \vee r_{\prec \mathbf{F}\, g} \right) $$

$$ \mathrm{enc}_n^3(\phi, k) = \left(\bigwedge_{\psi \in \Psi \setminus \mathsf{snfrule}_{\mathsf{L}}^{\mathbf{F}}} [\![\psi]\!]_k^k \right) \wedge \left(\bigwedge_{i=0}^{k} \bigwedge_{(f \Rightarrow \mathbf{F}\, g) \in \Psi} r_{\prec \mathbf{F}\, g} \rightarrow \bot \right) $$

$$\mathrm{enc}^3_l(\phi, k, l) = \left(\bigwedge_{\psi \in \Psi \setminus \mathsf{snfrule}^{\mathbf{F}}_{\mathsf{L}}} {}_l[\![\psi]\!]^k_k \right) \wedge \left(\bigwedge_{i=0}^{k} \bigwedge_{(f \Rightarrow \mathbf{F}\, g) \in \Psi} r_{\prec \mathbf{F}\, g} \rightarrow [\![\mathbf{F}\, g]\!]^l_k \right)$$

where $\Psi = T^*_{|}(\{\mathbf{start} \Rightarrow \textsc{Nnf}(\phi)\})$.

This encoding is not yet linear, however by a further factorisation we can achieve a linear encoding. Following the discussion in Clarke et al. [28], we introduce a set of new propositions $r^{*i}_{\mathbf{F}\, g}$ to rename each $[\![\mathbf{F}\, g]\!]^l_k$:

$$r^{*i}_{\mathbf{F}\, g} \rightarrow \begin{cases} [\![g \vee r^{*(i+1)}_{\mathbf{F}\, g}]\!]^i_k & \text{if } i < k \\ [\![g]\!]^i_k & \text{if } i = k \end{cases}$$

The correctness of this renaming follows from the usual arguments for propositional renaming (Section 2.2.1.1) and the positive polarity of the $\mathbf{F}\, g$.

This results in the following encoding which is linear size in the size of the original LTL:

$$\begin{aligned} \mathrm{enc}^4_c(\phi, k) = \bigwedge_{i=0}^{k-1} \Bigg(& \bigwedge_{\psi \in \Psi \setminus \mathsf{snfrule}^{\mathbf{F}}_{\mathsf{L}}} [\![\psi]\!]^i_k \\ & \wedge \bigwedge_{(f \Rightarrow \mathbf{F}\, g) \in \Psi} \left([\![f]\!]^i_k \rightarrow r^{*i}_{\mathbf{F}\, g} \vee r_{\prec \mathbf{F}\, g} \right) \wedge \left(r^{*i}_{\mathbf{F}\, g} \rightarrow [\![g]\!]^i_k \vee r^{*(i+1)}_{\mathbf{F}\, g} \right) \Bigg) \\ \wedge \bigwedge_{(f \Rightarrow \mathbf{F}\, g) \in \Psi} & \left([\![f]\!]^k_k \rightarrow r^{*k}_{\mathbf{F}\, g} \vee r_{\prec \mathbf{F}\, g} \right) \wedge \left(r^{*k}_{\mathbf{F}\, g} \rightarrow [\![g]\!]^k_k \right) \end{aligned}$$

$$\mathrm{enc}^4_n(\phi, k) = \left(\bigwedge_{\psi \in \Psi \setminus \mathsf{snfrule}^{\mathbf{F}}_{\mathsf{L}}} [\![\psi]\!]^k_k \right) \wedge \left(\bigwedge_{i=0}^{k} \bigwedge_{(f \Rightarrow \mathbf{F}\, g) \in \Psi} r_{\prec \mathbf{F}\, g} \rightarrow \bot \right)$$

$$\mathrm{enc}^4_l(\phi, k, l) = \left(\bigwedge_{\psi \in \Psi \setminus \mathsf{snfrule}^{\mathbf{F}}_{\mathsf{L}}} {}_l[\![\psi]\!]^k_k \right) \wedge \left(\bigwedge_{i=0}^{k} \bigwedge_{(f \Rightarrow \mathbf{F}\, g) \in \Psi} r_{\prec \mathbf{F}\, g} \rightarrow r^{*l}_{\mathbf{F}\, g} \right)$$

where $\Psi = T^*_{|}(\{\mathbf{start} \Rightarrow \textsc{Nnf}(\phi)\})$.

5.4 Further Optimisations

The additional changes to the encoding given below are not core observations about the encoding, and they do not change the asymptotic complexity. However, they improve or simplify the encoding to a useful extent.

5.4.1 SNF Variables

The conversion from LTL formulae to SNF involves the introduction of new existentially quantified variables. One additional cost of these variables, and the propositions that represent them, is the need for additional bi-implications to establish $\text{Loop}_{k,l}(\rho)$.

This additional cost in the encoding can be avoided, however. Figure 5.3a shows the infinite path represented by the propositions A and the variables Q if the loop constraint on the environment is used: the block, $A^1 \cup Q^1$–$A^4 \cup Q^4$, is repeated to extend the finite representation of the path to infinity. Conversely, if we omit the loop constraint on the environment (see Figure 5.3b), there is an additional set of variables available: Q^5 can differ from Q^1, although in both cases, the model state is the same (A^1). The nature of the loop means that the sequence of states repeated to infinity ($A^2 \cup Q^2$–$A^1 \cup Q^5$) is different from in the first case, although the cases differ only in the SNF variables.

We can see from the figures that simplifying the k-loop constraint affects only the sequence of states which are repeated. This does not imply a lengthening of the shortest witness path: since Q^1 and Q^5 may be identical, the shortest path may still be represented in the same way. Conversely, if the shortest witness path with the simplified constraint has $Q^1 \neq Q^5$ then this path may be shorter than for the standard case. Since the standard BMC encoding is known to produce the shortest possible witnesses and the basic PSNF encoding is equisatisfiable to the standard BMC encoding, there can be no such shorter witness.

In addition, we note that the main constraint on loopback positions comes from the model rather than the specification.

5.4.2 G in the k-Prefix Path Case

The SNF-based encodings attempt to minimise the differences between the k-prefix and k-loop path encodings, allowing for an increased amount of shared clauses between the two. One side-effect of this behaviour is a poor encoding for the **G** operator in the k-prefix case. In Section 3.1.1.1 we developed the sound semantics of LTL on k-prefix paths, and we observed that there is no witness possible for an LTL formula of the form $\mathbf{G}\,\phi$: the standard BMC

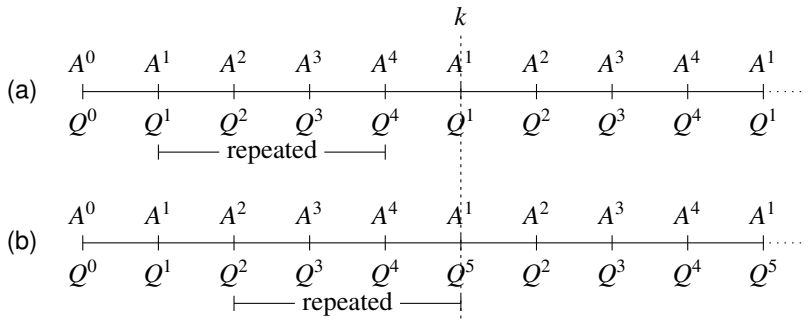


Figure 5.3: Illustration of loop lengths in a 5-1-loop path (a) with SNF variables included in the loop condition; (b) with SNF variables excluded from the loop condition

encoding in Table 3.1 gives the encoding of this expression as

$$[\![\mathbf{G}\,\phi]\!]^i_k \Leftrightarrow \bot$$

We wish to specialise the SNF encoding to take this into account, without compromising the level of sharing of other rules. That is, we want to assert that $P \Rightarrow \Phi[\mathbf{G}\,f]$ is encoded as $P \Rightarrow \Phi[f \wedge r_{\mathbf{X}\mathbf{G}\,f}]$ in the k-loop case, but $P \Rightarrow \Phi[\bot]$ in the k-prefix case. This can be achieved by changing the encoding of the other derived rule from the transformation of $\mathbf{G}$: $r_{\mathbf{X}\mathbf{G}\,f} \Rightarrow \mathbf{X}(f \wedge r_{\mathbf{X}\mathbf{G}\,f})$. Writing

$$r_{\mathbf{X}\mathbf{G}\,f} \Rightarrow \bot \wedge \mathbf{X}(f \wedge r_{\mathbf{X}\mathbf{G}\,f})$$

in the k-prefix case allows the factoring of the k-loop and k-prefix cases for $i < k$ to continue, but achieves the desired short-circuiting of the evaluation.

Lemma 5.8 (conjunctive X-rules in *k*-prefix paths) *In the context of a k-prefix path ϖ, the rules $f_0 \Rightarrow \mathbf{X}(f_0 \wedge f_1)$ and $f_0 \Rightarrow \bot \wedge \mathbf{X}(f_0 \wedge f_1)$ are equivalent.*

Proof We notice that $\varpi \models^k_k \mathbf{X}(f_0 \wedge f_1) \Leftrightarrow \bot$ and therefore $\varpi \models^k_k f_0 \Rightarrow \bot$. We use this as a base case to show by induction that for all $i < k$, $\varpi \models^i_k f_0 \Rightarrow \bot$, and hence the lemma holds by the usual rules of propositional logic. □

We use pattern matching in the definition of the encoding to identify the variables which were used in the fixpoint characterisations of **G** operators.

$$\mathrm{enc}^5_n(\phi,k) = \left(\bigwedge_{r_{\mathbf{X}\,\mathbf{G}\,\varphi}\in fv(\Psi)} r_{\mathbf{X}\,\mathbf{G}\,\varphi} \to \bot\right) \wedge \left(\bigwedge_{\psi\in\Psi\setminus \mathsf{snfrule}^{\mathbf{F}}_{\mathsf{L}}} [\![\psi]\!]^k_k\right) \wedge \left(\bigwedge_{i=0}^{k} \bigwedge_{(f\Rightarrow \mathbf{F}\,g)\in\Psi} r_{\prec \mathbf{F}\,g} \to \bot\right)$$

5.4.3 Variables and Eventualities

One of the steps in achieving the linear space encoding in Section 5.3.3 is defining additional propositions $r^{*i}_{\mathbf{F}\,g}$ each renaming $[\![\mathbf{F}\,g]\!]^i_k$. However, in the transformation step $T_{\mathbf{F}\uparrow}$, eventualities are brought to the top level of a rule's right hand side by a similar renaming, introducing the variable $r_{\mathbf{F}\,a}$ and hence the propositions $r^i_{\mathbf{F}\,a}$ in the encoding. This means that the part of $\mathrm{enc}^4_c(\phi,k)$ for a rule $r_{\mathbf{F}\,a} \Rightarrow \mathbf{F}\,a$ is

$$\left(\bigwedge_{i=0}^{k-1} \left(r^i_{\mathbf{F}\,a} \to r^{*i}_{\mathbf{F}\,a} \vee r_a\right) \wedge \left(r^{*i}_{\mathbf{F}\,a} \to [\![a]\!]^i_k \vee r^{*(i+1)}_{\mathbf{F}\,a}\right)\right) \wedge \left(r^k_{\mathbf{F}\,a} \to r^{*k}_{\mathbf{F}\,a} \vee r_{\prec \mathbf{F}\,g}\right) \wedge \left(r^{*k}_{\mathbf{F}\,a} \to [\![a]\!]^k_k\right)$$

The definition of renaming in Lemma 2.6 is $C[f] \cong (r_f \to f) \wedge C[r_f]$ for monotone context C. We can use this equisatisfiability to transform the above formula. We read it as a series of definitions of $r^i_{\mathbf{F}\,a}$, and so by multiple right-to-left applications of renaming we can replace every occurrence of $r^i_{\mathbf{F}\,a}$ by $r^{*i}_{\mathbf{F}\,a} \vee r_a$.

To write this replacement succinctly in the encoding we abbreviate the replacement of each defining proposition for eventualities,

$$\psi[r^{*0}_{\mathbf{F}\,a} \vee r_a / r^0_{\mathbf{F}\,a}] \ldots [r^{*k}_{\mathbf{F}\,a} \vee r_a / r^k_{\mathbf{F}\,a}][r^{*i}_{\mathbf{F}\,b} \vee r_b / r^i_{\mathbf{F}\,b}] \ldots$$

for each $r_{\mathbf{F}\,x} \in fv(T^*_{\uparrow}(\{\mathbf{start} \Rightarrow \mathrm{NNF}(\phi)\}))$ as $\psi[R]$. The encoding given below matches rules against a more precise pattern, $r_{\mathbf{F}\,g} \Rightarrow \mathbf{F}\,g$, to emphasise the identified variables.

$$\mathrm{enc}^6_c(\phi,k) = \bigwedge_{i=0}^{k-1} \left(\bigwedge_{\psi\in\Psi\setminus \mathsf{snfrule}^{\mathbf{F}}_{\mathsf{L}}} [\![\psi[R]]\!]^i_k \wedge \bigwedge_{(r_{\mathbf{F}\,g}\Rightarrow \mathbf{F}\,g)\in\Psi} \left(r^{*i}_{\mathbf{F}\,g} \to [\![g[R]]\!]^i_k \vee r^{*(i+1)}_{\mathbf{F}\,g}\right)\right) \wedge \bigwedge_{(r_{\mathbf{F}\,g}\Rightarrow \mathbf{F}\,g)\in\Psi} \left(r^{*k}_{\mathbf{F}\,g} \to [\![g[R]]\!]^k_k\right)$$

$$\mathrm{enc}_n^6(\phi, k) = \left(\bigwedge_{\psi \in \Psi \setminus \mathsf{snfrule}_{\mathsf{L}}^{\mathbf{F}}} [\![\psi[R]]\!]_k^k \right) \wedge \left(\bigwedge_{i=0}^{k} \bigwedge_{(r_{\mathbf{F}\,g} \Rightarrow \mathbf{F}\,g) \in \Psi} r_{\prec \mathbf{F}\,g} \rightarrow \bot \right)$$

$$\mathrm{enc}_l^6(\phi, k, l) = \left(\bigwedge_{\psi \in \Psi \setminus \mathsf{snfrule}_{\mathsf{L}}^{\mathbf{F}}} {}_l[\![\psi[R]]\!]_k^k \right) \wedge \left(\bigwedge_{i=0}^{k} \bigwedge_{(r_{\mathbf{F}\,g} \Rightarrow \mathbf{F}\,g) \in \Psi} r_{\prec \mathbf{F}\,g} \rightarrow r^{*l}_{\mathbf{F}\,g} \right)$$

where $\Psi = T^*_{\restriction}(\{\mathbf{start} \Rightarrow \mathrm{NNF}(\phi)\})$.

5.5 Related Work

There is no work directly related to that of this chapter, as the application of SNF to finite paths and then to BMC is entirely new. However, the approach of Latvala, Biere, Heljanko, and Junttila [71], described below, is related in several ways: they also give a linear-space encoding based on the fixpoint characterisations of temporal operators (as SNF is).

In *Simple Bounded LTL Model Checking*, Latvala, Biere, Heljanko, and Junttila [71] describe an alternative to the SNF approach to encoding which is nevertheless based on related principles. The encoding derived is, like SNF, linear in the size of the LTL formula and the bound.

The key development in Latvala et al. is the consideration in the loop case of two loop iterations. That is, for a loop path ab^ω, the encoding is defined over the states in path abb. As the model path is invariant between the two loop iterations, the same number of states and hence model variables are considered as in the standard BMC and SNF cases. The encoding is, like the SNF encoding, based on a consideration of the fixpoint characterisations of the LTL operators, although in form it has more in common with the procedure defined in Section 10.3.3. Unlike the SNF encoding, it is linear in size by construction—the approach described in Section 5.3.3 is not required. This makes the presentation more straightforward, and justifies the "simple" in the paper's title. On the other hand, the encoding is roughly double the size of the SNF encoding due to the doubling in the number of loop states considered.

A final key difference between the encodings is that the SNF encoding produces propositional logic very similar in form to CNF, the encoding of Latvala et al. is to a Boolean circuit. Although this makes it dependant on an efficient CNF conversion, it can also benefit from such a CNF conversion in a way that the SNF conversion cannot as it stands. Similarly, it may turn out to

be more suited to the Boolean circuit-based SAT solvers which are beginning to emerge at the time of writing.

Like the SNF work, this approach has been extended to encodings of past-time temporal logic [70].

5.6 Summary

In this chapter we have given a BMC encoding for PSNF. The basic encoding is based on a straightforward extension of BMC to allow for the existentially quantified variables introduced by the transformation to PSNF. This encoding, while straightforward to describe, is cubic with respect to k in the size of the result.

We therefore describe a series of improved encodings using an abstraction of the overall BMC formula into three parts: the common factors of the prefix and loop encodings are factored out into a common part, enc_c; the remaining parts of the prefix and loop encodings are in enc_n and enc_l. This allows us to take advantage of the similarity between the encodings for most of the rules (the main exceptions are **F**-rules) for most time steps (the encodings differ primarily at k).

The following improvements are made (index numbers correspond to the superscript indices of the encoding functions).

1. Basic factorisation of encoding at the rule level.

2. Improved treatment of eventualities using a similar approach to that of Clarke et al. [28] (see Section 3.4). This results in an encoding for eventualities that is similar to that resulting from the fixpoint characterisation of other operators. However, the transformation is entirely propositional and based on the renaming of repeated subformulae. The resulting encoding is quadratic size with respect to k.

3. Elimination of the dependency of the eventuality encoding on l in the loop case (this is the cause of the quadratic increase in size). This is based on the observation that an eventuality holds in a given state on a loop path if and only if it holds in the remaining states before k or in the states between the start of the loop and k. The encoding of eventualities

in the loop case is thus reduced to the the encoding in the prefix case. The second disjunct (states between the loop and k) is renamed by a single new proposition (not related to time) whose definition varies according to the position of the loop. This encoding is also quadratic but paves the way for the following encoding.

4. Linear encoding based on a combination of the previous two encodings. Since only the prefix encoding of eventualities is required due to the transformation of encoding 3, giving a new proposition for each eventuality at each state involves the introduction of only $k+1$ new proposition, each defined by a constant size formula in the common case.

The following further improvements consist of smaller changes to the encoding.

5. Interpretation of $\mathbf{G}$ as $\bot$ in the prefix case. Since the $\mathbf{G}$ operator has been eliminated by the transformation to SNF the encoding cannot take advantage of the fact that $\mathbf{G}\,\phi \equiv \bot$ in the prefix case. However, we identify the variables which are created during the fixpoint characterisation of $\mathbf{G}$ and assert that they are false in the prefix case.

6. Removal of redundant propositions in the linear encoding. Part of the linear encoding is to introduce new propositions for the interpretation of the eventualities in each state. However, propositions for this purpose already exist in the encoding as a consequence of the $T_{\mathbf{F}\uparrow}$ transformation, resulting in an unnecessary introduction of k propositions for each eventuality. These can be eliminated by noticing that renaming transformations can be applied in reverse, eliminating propositions and combining subformulae.

An additional improvement is to remove the explicit condition requiring the environment which gives values to variables to take the form of a loop, thus removing $O(k|\phi|)$ bi-implications from the resulting formula.

5.6.1 Transformation and Encoding

The final encoding developed above is summarised here. Given a specification $\phi \in |\mathrm{tl}|$ we convert it to SNF using the conversion function

$$T^*_{\uparrow} = \{T_{\mathbf{G}\uparrow}, T_{\mathbf{U}\uparrow}, T_{\mathbf{R}\uparrow}, T_{\mathbf{X}\uparrow}, T_{\mathbf{F}\uparrow}\}$$

The component transformations are given in Figure 5.6.1. The encoding is defined in terms of $\Psi = T^*_{\restriction}(\{\mathbf{start} \Rightarrow \mathrm{NNF}(\phi)\})$ by the following equations:

$$[\![\hat{M}]\!]_k \;\wedge\; \mathrm{enc}^6_c(\phi, k) \;\wedge\; \left(\mathrm{enc}^6_n(\phi, k) \vee \bigvee_{l<k} \left({}_l[\![\varpi(k) = \varpi(l)]\!]^0_k \wedge \mathrm{enc}^6_l(\phi, k, l)\right)\right)$$

where enc^6_c, enc^6_n, enc^6_l are given by

$$\mathrm{enc}^6_c(\phi, k) = \bigwedge_{i=0}^{k-1} \left(\bigwedge_{\psi \in \Psi \setminus \mathsf{snfrule}^{\mathbf{F}}_{\mathsf{L}}} [\![\psi[R]]\!]^i_k \wedge \bigwedge_{(r_{\mathbf{F}\,g} \Rightarrow \mathbf{F}\,g) \in \Psi} \left(r^{*i}_{\mathbf{F}\,g} \rightarrow [\![g[R]]\!]^i_k \vee r^{*(i+1)}_{\mathbf{F}\,g} \right) \right)$$
$$\wedge \bigwedge_{(r_{\mathbf{F}\,g} \Rightarrow \mathbf{F}\,g) \in \Psi} \left(r^{*k}_{\mathbf{F}\,g} \rightarrow [\![g[R]]\!]^k_k \right)$$

$$\mathrm{enc}^6_n(\phi, k) = \left(\bigwedge_{\psi \in \Psi \setminus \mathsf{snfrule}^{\mathbf{F}}_{\mathsf{L}}} [\![\psi[R]]\!]^k_k \right) \wedge \left(\bigwedge_{i=0}^{k} \bigwedge_{(r_{\mathbf{F}\,g} \Rightarrow \mathbf{F}\,g) \in \Psi} r_{\prec \mathbf{F}\,g} \rightarrow \bot \right)$$

$$\mathrm{enc}^6_l(\phi, k, l) = \left(\bigwedge_{\psi \in \Psi \setminus \mathsf{snfrule}^{\mathbf{F}}_{\mathsf{L}}} {}_l[\![\psi[R]]\!]^k_k \right) \wedge \left(\bigwedge_{i=0}^{k} \bigwedge_{(r_{\mathbf{F}\,g} \Rightarrow \mathbf{F}\,g) \in \Psi} r_{\prec \mathbf{F}\,g} \rightarrow r^{*l}_{\mathbf{F}\,g} \right)$$

The remaining encoding functions are given in Table 5.3.

$$T_{\mathbf{G}\uparrow}(P \Rightarrow \Phi[\mathbf{G}\, f]) \doteq \left\{ \begin{array}{l} P \Rightarrow \Phi\left[f \wedge r_{\mathbf{X}\mathbf{G}\, f}\right] \\ r_{\mathbf{X}\mathbf{G}\, f} \Rightarrow \mathbf{X}\left(f \wedge r_{\mathbf{X}\mathbf{G}\, f}\right) \end{array} \right\}$$

$$T_{\mathbf{U}\uparrow}(P \Rightarrow \Phi[f\, \mathbf{U}\, g]) \doteq \left\{ \begin{array}{l} P \Rightarrow \Phi\left[\left(g \vee \left(f \wedge r_{\mathbf{X}(f\, \mathbf{U}\, g)}\right)\right) \wedge \mathbf{F}\, g\right] \\ r_{\mathbf{X}(f\, \mathbf{U}\, g)} \Rightarrow \mathbf{X}(g \vee (f \wedge r_{\mathbf{X}(f\, \mathbf{U}\, g)})) \end{array} \right\}$$

$$T_{\mathbf{R}\uparrow}(P \Rightarrow \Phi[f\, \mathbf{R}\, g]) \doteq \left\{ \begin{array}{l} P \Rightarrow \Phi\left[g \wedge \left(f \vee r_{\mathbf{X}(f\, \mathbf{R}\, g)}\right)\right] \\ r_{\mathbf{X}(f\, \mathbf{R}\, g)} \Rightarrow \mathbf{X}(g \wedge (f \vee r_{\mathbf{X}(f\, \mathbf{R}\, g)})) \end{array} \right\}$$

$$T_{\mathbf{X}\uparrow}(P \Rightarrow \Phi^{+}[\mathbf{X}\, f]) \doteq \left\{ \begin{array}{l} P \Rightarrow \Phi^{+}\left[r_{\mathbf{X}\, f}\right] \\ r_{\mathbf{X}\, f} \Rightarrow \mathbf{X}\, f \end{array} \right\}$$

Figure 5.4: The SNF transformation functions for BMC

Table 5.3: The BMC encoding for PSNF rules

ϕ	$[\![\phi]\!]^i_k$	${}_l[\![\phi]\!]^i_k$
a	a^i	a^i
α	α^i	α^i
$\neg\phi$	$\neg\, [\![\phi]\!]^k_i$	$\neg\, {}_l[\![\phi]\!]^k_i$
$\phi \wedge \psi$	$[\![\phi]\!]^i_k \wedge [\![\psi]\!]^i_k$	${}_l[\![\phi]\!]^i_k \wedge {}_l[\![\psi]\!]^i_k$
$\phi \vee \psi$	$[\![\phi]\!]^i_k \vee {}_l[\![\psi]\!]^i_k$	${}_l[\![\phi]\!]^i_k \vee {}_l[\![\psi]\!]^i_k$
$\mathbf{start} \Rightarrow f$	$i > 0 \vee [\![f]\!]^i_k$	$i > 0 \vee {}_l[\![f]\!]$
$\mathbf{start} \Rightarrow \mathbf{X}\, f$	$i > 0 \vee [\![f]\!]^1_k$	$i > 0 \vee {}_l[\![f]\!]^1$
$f \Rightarrow g$	$[\![f]\!]^i_k \rightarrow [\![g]\!]^i_k$	${}_l[\![f]\!]^i_k \rightarrow {}_l[\![f]\!]^i_k$
$f \Rightarrow \mathbf{X}\, g$	$[\![f]\!]^i_k \rightarrow (i < k \wedge [\![g]\!]^{i+1}_k)$	${}_l[\![f]\!]^i_k \rightarrow \left(i < k \wedge {}_l[\![f]\!]^{i+1}_k\right) \vee \left(i = k \wedge {}_l[\![f]\!]^l_k\right)$

Chapter 6

CNF Conversion of RBCs

Reduced Boolean circuits (see Section 2.3.1) are a convenient representation of propositional formulae as they help to identify subformulae which occur multiple times. As part of achieving this, the set of operations is reduced in two ways: they are restricted to $\wedge$ and $\leftrightarrow$, and they must have exactly two arguments. From the point of view of CNF conversion, this latter restriction has a profound impact: the usual definitional conversion used (Section 2.3.1.2) introduces a variable for each operator. Where an n-ary operator in the original formula has been converted to a structure of $n-1$ binary operators for the RBC representation, the number of variables and clauses needed in CNF is similarly increased.

A particularly notable case is the RBC representation of a CNF formula: for a structure with n leaves, up to $n-1$ internal vertices may be required. A CNF formula consisting of n clauses of m literals inserted into an RBC will, after CNF conversion, produce a formula with $nm-1$ additional literals and a total of nm clauses. The SNF encoding for BMC described in Chapter 5 produces propositional logic which is very close to clause form, so the use of the definitional clause form conversion in this situation is extremely detrimental to the performance of BMC using this encoding.

To improve the CNF conversion of RBCs, we investigate the application to RBCs of the improved clause form conversions summarised in Section 2.2.1.1. We extend this work to the clause form conversion proposed by Boy de la Tour [17] which, for certain classes of input formula, is optimal in the number of clauses. The main contribution of this chapter, however, is a new clause form conversion with the output size advantages of the Boy

de la Tour conversion (optimality in the same cases) but with linear rather than quadratic time complexity. The restrictions of RBCs simplify the definition of this conversion and the presentation of the optimality result significantly.

The ideas in this chapter were first presented as a short paper at the Conference on the Theory and Applications of Satisfiability Checking (SAT) 2004 in Vancouver, Canada [90]; the improved presentation here, the result of joint work with Paul Jackson, was accepted into the post-conference proceedings [63].

6.1 CNF Conversions on Linear Trees

We begin by examining CNF conversions for certain restrictions of RBCs. These will become building blocks for the CNF conversions of full RBCs. Tree RBCs allow us to ignore the possibility of shared vertices; linear trees are a stronger restriction representing linear formulae (those without equivalence operators) without taking into account the possibility for sharing.

Definition 6.1.1 (tree RBC)
A *tree RBC* is an RBC in which no vertices are shared: the graph is a tree.

Definition 6.1.2 (linear tree)
A *linear tree* is a tree RBC in which every internal vertex is a conjunction.

Given a tree RBC, we define an additional property of vertices to allow us to directly refer to its incoming edge. We also extend edges with a source property and the ability to identify edges which share a common parent vertex. This means that we can write functions to traverse linear trees bottom-up as well as top-down.

$$inedge(V) = E \qquad \text{where } target(E) = V$$

$$source(E) = \begin{cases} V & \text{if } E = left(V) \vee E = right(V) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$sib(E) = \begin{cases} left(source(E)) & \text{if } E = right(source(E)) \\ right(source(E)) & \text{if } E = left(source(E)) \end{cases}$$

To simplify the definition of the CNF conversions we adopt the set-theoretic notation for clause form described in Section 2.2.1. This means that we can make use of the union ($\cup$) operator, combining two sets of clauses together, and the cross-multiply operator ($\times$), which forms the set of clauses corresponding to the application of the distributive rule for $\wedge$ over $\vee$ (see Section 2.1).

Definition 6.1.3 (cross-multiply)

The *cross-multiply* operation on sets of sets is given by

$$C_1 \times C_2 = \{x \cup y \mid x \in C_1, y \in C_2\}$$

Cross-multiply applied to CNF sets-of-sets corresponds to the standard CNF conversion of the disjunction of the two sets:

$$C_1 \times C_2 = \textsc{Cnf}\left(\left(\bigwedge_{c \in C_1} \bigvee_{l \in c} l\right) \vee \left(\bigwedge_{c \in C_2} \bigvee_{l \in c} l\right)\right)$$

We use the standard notation $|C|$ to refer to the number of clauses in set C.

6.1.1 The Standard and Definitional Conversions

The *standard* CNF conversion (Figure 2.2.1) is obtained for propositional formulae by the repeated application of the distributive rule $f_0 \vee (f_1 \wedge f_2) \equiv (f_0 \vee f_1) \wedge (f_0 \vee f_2)$ to a formula already in NNF. The restrictions on RBCs means that this is not directly applicable—NNF is not representable as an RBC, so the intermediate products of CNF are not representable. However, we can write the conversion as a function from RBCs to sets of clauses provided some care is taken in handling disjunctions.

The notion of polarity as given in Section 2.2.1.2 can be extended to vertices of an RBC in the case that the RBC is a tree, although in this case we must give a more functional definition than Definition 2.2.11.

Definition 6.1.4 (polarity for RBCs)

The polarity of a vertex V in a tree RBC (an RBC with no shared vertices) is given with respect to a root vertex or edge T by the expression $\mathrm{pol}(T, V)$ (defined below); the resulting value is 1, -1, or 0, corresponding to a positive, negative or zero polarity.

$$\mathrm{pol}(T, T) = 1$$

$$\mathrm{pol}(T, V) = \begin{cases} \mathrm{pol}(T, \mathit{inedge}(V)) & \text{if } \mathit{sign}(\mathit{inedge}(V)) = + \\ -\mathrm{pol}(T, \mathit{inedge}(V)) & \text{if } \mathit{sign}(\mathit{inedge}(V)) = - \end{cases}$$

$$\mathrm{pol}(T, E) = \begin{cases} 0 & \text{if } op(source(E)) = \leftrightarrow, \text{ or} \\ \mathrm{pol}(T, source(E)) & \text{otherwise} \end{cases}$$

This definition of polarity can be generalised to non-tree RBCs by first defining a generalisation of *inedge* for the shared-vertex case; we do not complicate the definition above with this as it is not useful to the development of this chapter.

The polarity of conjunction vertices determines whether, if converted to NNF, they would be interpreted as disjunctions. By checking polarity we can avoid the explicit conversion to NNF.

Four mutually recursive functions, given in Figure 6.1, define the standard clause form conversion for tree RBCs. $\textsc{Cnf}(V)$ converts a vertex V and its descendant tree to clause form in the case that V has positive polarity by forming the conjunction of the two descendant sub-trees; $\textsc{Cnf}^{-}(V)$ gives the corresponding conversion if V has negative polarity by forming the disjunction (using the cross-multiply operation) of the two descendant sub-trees. The polarity of each vertex is determined by its incoming edge: $\textsc{Cnf}(E)$, used when its parent vertex has positive polarity, converts an edge E and its descendant tree to clause form by determining the sign of the edge and using $\textsc{Cnf}(V)$ or $\textsc{Cnf}^{-}(V)$ on the target of the edge as appropriate. Similarly, $\textsc{Cnf}^{-}(E)$ is used if the parent vertex of the edge has negative polarity.

We give a function to obtain the definitional clause form conversion for RBCs in Section 2.3.1.2.

6.1.2 Polarity-Dependant Renaming Conversions

We noted in Section 2.2.1.1 that the definitional conversion could be refined by the consideration of subformula polarity [84]. The structure-preserving clause form conversion was presented in Section 2.2.1.1 as a preprocessing step, converting a formula f into an f' which converts to a more compact clause form using the standard conversion. The corresponding conversion for RBCs is that which constructs a new RBC from an old one, taking into account the polarity of each vertex in order to construct the appropriate implication. This is given in Figure 6.2. Notice that we use the pol function rather than using function parameters to track polarity (as in Figures 2.5 and 6.1).

To simplify the definitions, we introduce two new constructor functions, $\mathrm{L}(s, r)$, for constructing a new leaf vertex with an incoming edge, and $\mathrm{E}(s, V)$

$$\textsc{Cnf}(E) = \begin{cases} \textsc{Cnf}(target(E)) & \text{if } sign(E) = + \\ \textsc{Cnf}^{-}(target(E)) & \text{if } sign(E) = - \end{cases}$$

$$\textsc{Cnf}^{-}(E) = \begin{cases} \textsc{Cnf}^{-}(target(V)) & \text{if } sign(E) = + \\ \textsc{Cnf}(target(V)) & \text{if } sign(E) = - \end{cases}$$

$$\textsc{Cnf}(V) = \begin{cases} \{\{var(V)\}\} & \text{if } V \in \mathbf{V_L}, \text{ or} \\ \textsc{Cnf}(left(V)) \cup \textsc{Cnf}(right(V)) & \text{if } op(V) = \wedge, \text{ or} \\ \begin{array}{l} (\textsc{Cnf}(left(V)) \times \textsc{Cnf}^{-}(right(V))) \\ \cup\, (\textsc{Cnf}^{-}(left(V)) \times \textsc{Cnf}(right(V))) \end{array} & \text{if } op(V) = \leftrightarrow \end{cases}$$

$$\textsc{Cnf}^{-}(V) = \begin{cases} \{\{\neg\, var(V)\}\} & \text{if } V \in \mathbf{V_L}, \text{ or} \\ \textsc{Cnf}^{-}(left(V)) \times \textsc{Cnf}^{-}(right(V)) & \text{if } op(V) = \wedge, \text{ or} \\ \begin{array}{l} (\textsc{Cnf}(left(V)) \times \textsc{Cnf}(right(V))) \\ \cup\, (\textsc{Cnf}^{-}(left(V)) \times \textsc{Cnf}^{-}(right(V))) \end{array} & \text{if } op(V) = \leftrightarrow \end{cases}$$

Figure 6.1: The standard clause form conversion for tree RBCs

$$\mathrm{SP}(T) = rbc(\mathrm{L}(+, r_T), \mathrm{SP}_R(T, T), \wedge, +)$$

$$\mathrm{SP}_R(T, E) = \mathrm{SP}_R(T, target(E))$$

$$\mathrm{SP}_R(T, V) = \begin{cases} \top & \text{if } V \in \mathbf{V_L} \\ rbc(\left\{ \begin{array}{ll} rbc(\mathrm{L}(+, r_V), \mathrm{sub}^-(V), \wedge, -) & \text{if } \mathrm{pol}(T, V) = 1 \\ rbc(\mathrm{L}(-, r_V), \mathrm{sub}^+(V), \wedge, -) & \text{if } \mathrm{pol}(T, V) = -1 \\ rbc(\mathrm{L}(+, r_V), \mathrm{sub}^+(V), \leftrightarrow, +) & \text{if } \mathrm{pol}(T, V) = 0 \end{array} \right\}, & \\ \quad rbc(\mathrm{SP}_R(T, left(V)), \mathrm{SP}_R(T, right(V)), \wedge, +), & \\ \quad \wedge, +) & \text{if } V \in \mathbf{V_I} \end{cases}$$

$$\mathrm{sub}(E) = \mathrm{L}(sign(E), r_{target(E)})$$

$$\mathrm{sub}^s(V) = rbc(\mathrm{sub}(left(V)), \mathrm{sub}(right(V)), \wedge, s)$$

Figure 6.2: The structure-preserving renaming construction $\mathrm{SP}(T)$. Function $\mathrm{sub}(T)$ constructs the graph with root edge T with renamed subgraphs replaced by variables

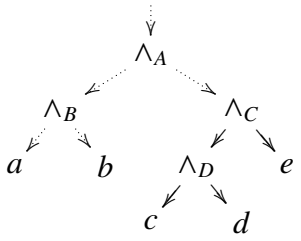
for constructing a new edge connected to vertex V:

$$\mathrm{L}(s, r) = E \qquad \text{where } target(E) = V, sign(E) = s \text{ and } var(V) = r$$
$$\mathrm{E}(s, V) = E \qquad \text{where } target(E) = V \text{ and } sign(E) = s$$

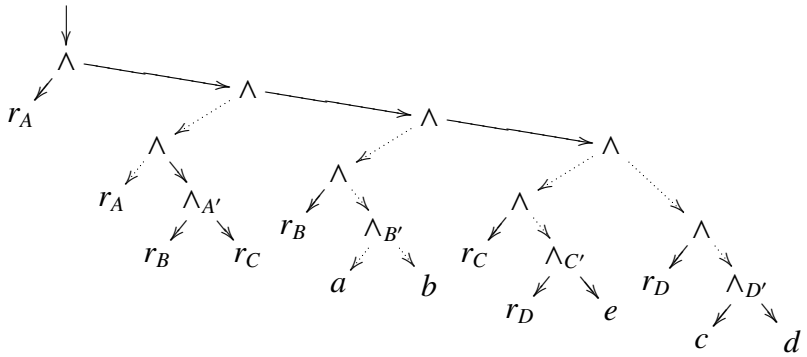
The structure of the conversion function is such that SP_R recurses down the tree and returns a chain of conjunctions of definitions of subformulae. The auxiliary function sub constructs the defined subformula itself, as a conjunction of appropriate sign of the variables representing the appropriate subgraphs. Figure 6.3b shows the result of applying SP to the tree in Figure 6.3a.

The structure-preserving conversion is representative of a larger class of conversions: rather than renaming every vertex, we can consider any equisatisfiable transformation on the formula. We specifically consider the generalisation of the SP conversion to selectively renaming only a subset of vertices.

For tree RBCs, we consider only renamings of *vertices* (other analyses place an equivalent restriction forbidding the renaming of subformulae with negation as the main connective). The order in which renamings are made does



(a) $(a \vee b) \rightarrow (a \wedge b \wedge c)$



(b) After application of SP

Figure 6.3: Example of the application of the structure-preserving CNF conversion to an RBC

not affect the final result due to the commutativity of $\wedge$, so we are able to give renaming-based clause form conversions in terms of the sets of vertices that they rename. The transformation in Figure 6.4 constructs a graph consisting of the renamed formula and the subgraph defining constraints on the new variables. This is sufficient to allow us to write the structure-preserving clause form conversion as

$$\mathrm{SP}(T) = \textsc{Cnf}(\mathrm{ren}(T, \mathbf{V_I}))$$

As we noted in Section 2.2.1, the standard and SP conversions both have their drawbacks characterised by particular types of original formulae (DNF formulae for the standard conversion and CNF formulae for the SP conversion). The renaming set approach is a way of combining both procedures; the construction of the renaming set must be such that the drawbacks of the two procedures are eliminated.

6.1.3 The Conversion due to Boy de la Tour

Boy de la Tour [18] presents a comprehensive solution to the problem of

$$\mathrm{ren}(T, \mathbf{R}) = rbc(\mathrm{def}(T, T, \mathbf{R}), \mathrm{sub}(T, \mathbf{R}), \wedge, +)$$

$$\mathrm{def}(T, E, \mathbf{R}) = \mathrm{def}(T, target(E), \mathbf{R})$$

$$\mathrm{def}(T, V, \mathbf{R}) = \begin{cases} \mathrm{L}(+, \top) & \text{if } V \in \mathbf{V_L} \\ rbc(\left\{ \begin{array}{ll} \mathrm{L}(+, \top) & \text{if } V \notin \mathbf{R} \\ rbc(\mathrm{L}(+, r_V), \mathrm{sub}^-(V, \mathbf{R} \setminus \{V\}), \wedge, -) & \text{if } \mathrm{pol}(T, V) = 1 \\ rbc(\mathrm{L}(-, r_V), \mathrm{sub}^+(V, \mathbf{R} \setminus \{V\}), \wedge, -) & \text{if } \mathrm{pol}(T, V) = -1 \end{array} \right\}, & \\ \quad rbc(\mathrm{def}(T, left(V), \mathbf{R}), \mathrm{def}(T, right(V), \mathbf{R}), \wedge, +), & \\ \quad \wedge, +) & \text{if } V \in \mathbf{V_I} \end{cases}$$

$$\mathrm{sub}(E, \mathbf{R}) = \mathrm{sub}^{sign(E)}(target(E), \mathbf{R})$$

$$\mathrm{sub}^s(V, \mathbf{R}) = \begin{cases} \mathrm{E}(s, V) & \text{if } V \in \mathbf{V_L} \\ \mathrm{L}(s, r_V) & \text{if } V \in \mathbf{R} \\ rbc(\mathrm{sub}(left(V), \mathbf{R}), \mathrm{sub}(right(V), \mathbf{R}), \wedge, s) & \text{otherwise,} \end{cases}$$

Figure 6.4: The vertex-based renaming construction $\mathrm{ren}(T, \mathbf{R})$. Function $\mathrm{sub}(T, \mathbf{R})$ returns the graph with root edge T with renamed subgraphs replaced by variables; $\mathrm{def}(T, T', \mathbf{R})$ returns the graph defining all the introduced variables below T' with respect to root T

Table 6.1: The clause counting functions $p^+(V)$ and $p^-(V)$

	$p^+(E)$	$p^-(E)$
$sign(E) = +$	$p^+(target(E))$	$p^-(target(E))$
$sign(E) = -$	$p^-(target(E))$	$p^+(target(E))$

	$p^+(V)$	$p^-(V)$
$v \in \mathbf{V_L}$	1	1
$op(V) = \wedge$	$p^+(left(V)) + p^+(right(V))$	$p^-(left(V)) \cdot p^-(right(V))$
$op(V) = \leftrightarrow$	$p^+(left(V))p^-(right(V)) + p^-(left(V))p^+(right(V))$	$p^+(left(V))p^+(right(V)) + p^-(left(V))p^-(right(V))$

choosing the subformulae to rename. The approach taken is to compute the impact of renaming any given subformula and to perform the renaming only if it will not increase the number of clauses produced by the formula as a whole. The conversion is shown to be optimal for formulae without equivalences, and we will make use of this property in order to prove the optimality of the new conversion in Section 6.2.

The adaptation to tree RBCs presented below removes some of the ambiguities from the original presentation as we can refer uniquely to vertices in the tree rather than to subformulae.

Boy de la Tour defines the functions $p^+(T) = |\textsc{Cnf}(T)|$ and $p^-(T) = |\textsc{Cnf}(\neg T)|$, counting the number of clauses in the positive and negative clause forms for T, using a simple look-up table (Table 6.1) which enables these values to be computed without constructing the clauses themselves. The *benefit* (that is, the reduction in the total number of clauses) of renaming a vertex V in a tree T is given by

$$B(T, V) = p^+(T) - p^+(\text{ren}(T, \{V\}))$$

In order to make a decision about renaming at a particular vertex without needing to analyse the whole tree, $p^+(T)$ is rewritten in terms of $p^+(V)$ and $p^-(V)$:

$$p^+(T) = a_V^T p^+(V) + b_V^T p^-(V) + c_V^T$$

Table 6.2: Computation of the coefficients a_V^T and b_V^T

	a_V^T	b_V^T
$V = T$	1	0
$sign(inedge(V)) = +$	$a^T_{inedge(V)}$	$b^T_{inedge(V)}$
$sign(inedge(V)) = -$	$b^T_{inedge(V)}$	$a^T_{inedge(V)}$

	a_E^T	b_E^T
$E = T$	1	0
$op(target(E)) = \wedge$	$a^T_{source(E)}$	$b^T_{source(E)} p^-(sib\, E)$
$op(target(E)) = \leftrightarrow$	$a^T_{source(V)} p^-(sib(E)) +$ $b^T_{inedge(V)} p^+(sib(E))$	$a^T_{source(V)} p^+(sib(E)) +$ $b^T_{inedge(V)} p^-(sib(E))$

Where the coefficients a and b may be considered as the number of occurrences of the clauses representing V and $\neg V$ respectively, such that the first sum counts the total number of clauses including subformulae of V; the coefficient c represents the number of clauses due to the rest of the tree. a and b are computed from the context of V as in Table 6.2. Note that the values are related to the polarity of the vertices: $a_V^T = 0$ if $\text{pol}(T, V) = -1$ and $b_V^T = 0$ if $\text{pol}(T, V) = 1$. When computing the benefit, the coefficient c is cancelled, so we do not need to give its construction. The benefit function can now be given in terms of polarity as

$$B(T,V) = \begin{cases} a_V^T p^+(V) \;-\; (a_V^T + p^+(V)) & \text{if } \text{pol}(T,V) = 1 \\ b_V^T p^-(V) \;-\; (b_V^T + p^-(V)) & \text{if } \text{pol}(T,V) = -1 \\ a_V^T p^+(V) + b_V^T p^-(V) \\ \qquad\qquad -\; (a_V^T + b_V^T + p^+(V) + p^-(V)) & \text{if } \text{pol}(T,V) = 0 \end{cases}$$

The algorithm given by Boy de la Tour is a top-down computation of the benefit of a renaming given the renamings that have gone before. Boy de la Tour makes use of annotations on the subformulae to store computed values of p^+ and p^-; these are updated as renamings occur in order to reflect the changing size of subformulae encodings. To enable a purely functional definition of the renaming set construction, we define a new pair of clause-counting functions

Table 6.3: The renaming-compensated clause counting functions $p_r^+(T, \mathbf{R})$ and $p_r^-(T, \mathbf{R})$

	$p_r^+(E, \mathbf{R})$	$p_r^-(E, \mathbf{R})$
$sign(E) = +$	$p_r^+(target(E), \mathbf{R})$	$p_r^-(target(E), \mathbf{R})$
$sign(E) = -$	$p_r^-(target(E), \mathbf{R})$	$p_r^+(target(E), \mathbf{R})$

	$p_r^+(V, \mathbf{R})$	$p_r^-(V, \mathbf{R})$
$V \in \mathbf{V_L}$	1	1
$V \in \mathbf{R}$	1	1
$op(V) = \wedge$	$p_r^+(left(V), \mathbf{R})$ $+p_r^+(right(V), \mathbf{R})$	$p_r^-(left(V), \mathbf{R})$ $\cdot p_r^-(right(V), \mathbf{R})$

$p_r^+(V, \mathbf{R})$ and $p_r^-(V, \mathbf{R})$ which count the number of clauses produced by the graph beginning at vertex V after the application of renaming $\mathbf{R}$ (Table 6.3). That is, $p_r^s(V, \mathbf{R}) = |\text{sub}^s(V, \mathbf{R})|$ (the clauses in $\text{def}^s(V, \mathbf{R})$ are disregarded as they play no further part in determining the size of the result). The benefit function is redefined to take into account the current renaming set:

$$B(T, V, \mathbf{R}) = \begin{cases} a_V^{\text{ren}(T,\mathbf{R})} p_r^+(V, \mathbf{R}) \; - \; (a_V^{\text{ren}(T,\mathbf{R})} + p_r^+(V, \mathbf{R})) & \text{if pol}(T, V) = 1 \\ b_V^{\text{ren}(T,\mathbf{R})} p_r^-(V, \mathbf{R}) \; - \; (b_V^{\text{ren}(T,\mathbf{R})} + p_r^-(V, \mathbf{R})) & \text{if pol}(T, V) = -1 \\ a_V^{\text{ren}(T,\mathbf{R})} p_r^+(V, \mathbf{R}) + b_V^{\text{ren}(T,\mathbf{R})} p_r^-(V, \mathbf{R}) - & \\ (a_V^{\text{ren}(T,\mathbf{R})} + b_V^{\text{ren}(T,\mathbf{R})} + p_r^+(V, \mathbf{R}) + p_r^-(V, \mathbf{R})) & \text{if pol}(T, V) = 0 \end{cases}$$

We give the construction of the renaming set in Figure 6.5 allowing us to write the algorithm as

$$\text{BDLT}(T) = \text{CNF}(\text{ren}(T, \text{BDLT}(T, T, \emptyset)))$$

The main drawback of this approach is that the value of the functions p^+ and p^- can grow exponentially with the depth of the tree; precomputing the values as suggested by Boy de la Tour is likely to be unimplementable because of the large numbers involved: examining these functions in relation to BMC [91] demonstrates that even trivial examples will overflow a 32-bit register.

$$\mathrm{BDLT}(T, E, \mathbf{R}) = \begin{cases} \mathrm{BDLT}(T, \mathit{target}(V), \mathbf{R}) & \text{if } \mathit{sign}(E) = + \\ \mathrm{BDLT}(T, \mathit{target}(V), \mathbf{R}) & \text{if } \mathit{sign}(E) = - \end{cases}$$

$$\mathrm{BDLT}(T, V, \mathbf{R}) = \begin{cases} \mathbf{R} & \text{if } V \in \mathbf{V_L}, \text{ or} \\ \mathrm{BDLT}(T, \mathit{right}(V), & \\ \qquad \mathrm{BDLT}(T, \mathit{left}(V), \mathbf{R})) & \text{if } B(T, V, \mathbf{R}) < 0 \\ \mathrm{BDLT}(T, \mathit{right}(V), & \\ \qquad \mathrm{BDLT}(T, \mathit{left}(V), \mathbf{R} \cup \{V\})) & \text{if } B(T, V, \mathbf{R}) \geq 0 \end{cases}$$

Figure 6.5: Renaming sets construction for the Boy de la Tour conversion

Boy de la Tour [18] describes a dynamic programming implementation of $\mathrm{BDLT}(T, V, \mathbf{R})$ which improves performance by recording the values of p^+ and p^- at each vertex and keeping track of the current values of coefficients a and b during the traversal of the tree. This means that $B(T, V, \mathbf{R})$ requires only $O(1)$ computations at each vertex, but the arithmetic must be done on $|\mathbf{V}|$-bit words[1] which leads to a per-vertex complexity of $O(|\mathbf{V}|)$. The resulting algorithm is $O(|\mathbf{V}|^2)$ in contrast to DEF and SP which are both linear in the number of vertices.

A more recent presentation of the algorithm by Nonnengart, Rock, and Weidenbach [81] removes the requirement for arbitrary-length arithmetic by reducing the test $B(T, V, \mathbf{R}) \geq 0$ to a number of case splits. For example, if V has positive polarity, the condition for renaming is

$$p_r^+(V, \mathbf{R}) > 1 \text{ and } a_V^{\mathrm{ren}(T,\mathbf{R})} > 1$$

This can be checked efficiently: $p_r^+(V, \mathbf{R}) > 1$ can be determined by searching the subgraph below V for equivalences or conjunctions with positive polarity; similar conditions exist for $a_V^{\mathrm{ren}(T,\mathbf{R})}$. Unfortunately, these become quite elaborate, especially for zero polarity formulae. In that case, there are eight different syntactic conditions in various combinations. The paper [81] does

[1] Boy de la Tour uses $O(n)$ as the time complexity of n-bit multiplication, but does not explain the origin of that figure.

not give the detail of these—the full syntactic conditions are given only in the source code of the theorem prover SPASS [100]. One of the motivations for devising the CNF conversion described below is the relative complexities both of arbitrary-length arithmetic and of the Nonnengart et al. conditions.

6.2 The Compact Conversion

We present a new clause form conversion, the *compact* conversion[2] defined first on linear trees, then extended to general RBCs. It computes the sets of renaming locally and bottom-up.

For each vertex we consider the number of clauses it will generate based on whether a child vertex is renamed. Consider a disjunction $f_0 \vee f_1$; we assume that the renaming process has already been completed on all of the subformulae of f_0 and f_1 as appropriate. Then the disjunction is converted by either

- renaming one argument, eg f_0 to r_{f_0}; this produces the definition $r_{f_0} \rightarrow f_0$, and replaces the original disjunction with the renamed form $r_{f_0} \vee f_1$; or
- computing the disjunction according to the standard CNF conversion, ultimately producing $\textsc{Cnf}(f_0) \times \textsc{Cnf}(f_1)$.

The key idea of the compact conversion is to make the decision between these two options not by considering the impact on the formula as a whole, as Boy de la Tour does, but instead by computing the number of clauses that will eventually be used to represent the disjunction. This number is equal to the sum or the product of the number of clauses in f_0 and f_1.

More precisely, we define the function $\textsc{Comp}(T, V)$ in Figure 6.6 to give the set of renamings on the tree beginning at V. The auxiliary function $\mathrm{dis}(V)$ chooses the best child of V, if any, to rename by using the sum-versus-product decision. The renaming condition is computed on the tree after all vertices below the considered one have been renamed.

The compact conversion is given by

$$\textsc{Comp}(T) = \textsc{Cnf}(\mathrm{ren}(T, \textsc{Comp}(T, T)))$$

[2]The name is chosen to indicate that the conversion is compact both in its *output* (number of clauses with respect to the standard and structure-preserving conversions) and its *implementation* (with respect to the Nonnengart et al. implementation of the Boy de la Tour conversion).

$$\textsc{Comp}(T, E) = \textsc{Comp}(T, \mathit{target}(V))$$

$$\textsc{Comp}(T, V) = \begin{cases} \emptyset & \text{if } V \in \mathbf{V_L}, \text{ or} \\ \textsc{Comp}(T, \mathit{left}(V)) \cup \textsc{Comp}(T, \mathit{right}(V)) & \text{if } \mathrm{pol}(T, V) = 1, \text{ or} \\ \mathrm{dis}(V) \cup \textsc{Comp}(T, \mathit{left}(V)) \cup \textsc{Comp}(T, \mathit{right}(V)) & \text{if } \mathrm{pol}(T, V) = -1 \end{cases}$$

$$\mathrm{dis}(V) = \left\{ \begin{array}{ll} \emptyset & \text{if } n_l n_r \leq n_l + n_r, \text{ or} \\ \{\mathit{target}(\mathit{left}(V))\} & \text{if } n_l > n_r \\ \{\mathit{target}(\mathit{right}(V))\} & \text{if } n_l \leq n_r \end{array} \right\}$$

$$\text{where } \left\{ \begin{array}{l} n_l = p_r^-(\mathit{left}(V), \textsc{Comp}(T, \mathit{left}(V))) \\ n_r = p_r^-(\mathit{right}(V), \textsc{Comp}(T, \mathit{right}(V))) \end{array} \right\}$$

Figure 6.6: Renaming sets construction for the compact conversion

Since we are targeting a SAT solver with this conversion, with its (assumed) exponential complexity in the number of variables, we choose to rename only if it *reduces* the number of clauses produced. In the case that the number of clauses is the same, the renaming is not performed. This is in contrast to the Boy de la Tour conversion, where the optimality analysis is simplified by the zero-benefit renaming.

The correctness of the compact conversion algorithm follows from the correctness of renaming (see Lemma 2.7) on arbitrary renaming sets.

6.3 Optimality of the Compact Conversion for Linear Trees

We show the optimality of the compact conversion by a comparison with the Boy de la Tour conversion. We establish which vertices appear in the renaming sets of one conversion and not the other, and then analyse the impact that the differences make.

When comparing the decision taken to include a vertex in the renaming sets by the two algorithms we take into account the different contexts: in the

Boy de la Tour algorithm, the superformulae and left sibling subtree have already been renamed; in the compact conversion the subformulae have been renamed. Writing $\mathbf{R}$ for a set of renamings, we have $\mathbf{R}_{\sqsupset V}$ for the subset of renamings involving the superformulae and left sibling subtree (if it exists) of V and $\mathbf{R}_{\sqsubset V}$ for the subset involving the subformulae of V.

The decisions made by the compact conversion take into account only p_r^+ and p_r^-—but these are computed after subformula renaming. That is, the decision to rename vertex V_1 in $V_1 \wedge V_2$ is based on the values $p_r^+(V_1, \mathbf{R}_{\sqsubset V_1})$, $p_r^+(V_2, \mathbf{R}_{\sqsubset V_2})$ and their complements. In contrast, for the Boy de la Tour algorithm the same decision is made by considering the values $a_{V_1}^{\mathrm{ren}(T,\mathbf{R}_{\sqsupset V_1})}$, $b_{V_1}^{\mathrm{ren}(T,\mathbf{R}_{\sqsupset V_1})}$, $p_r^+(V_1, \mathbf{R}_{\sqsupset V_1})$, and $p_r^-(V_1, \mathbf{R}_{\sqsupset V_1})$. We begin by establishing some basic lemmas about the Boy de la Tour coefficients a and b and the clause counting functions p^s and p_r^s.

Lemma 6.1 (coefficient values under renaming) *For a vertex V and renaming $\mathbf{R}$ on tree T with $V \in \mathbf{R}$,*

$$a_V^{\mathrm{ren}(T,\mathbf{R})} = 1 \text{ if } \mathrm{pol}(T, V) = 1, \text{ and}$$
$$b_V^{\mathrm{ren}(T,\mathbf{R})} = 1 \text{ if } \mathrm{pol}(T, V) = -1$$

Proof After renaming, a vertex V becomes part of the definition of the replacement variable r_V. As shown in Figure 6.4, the definition is attached by a tree of positive conjunctions to the root and the sign of the inedge of V reflecting its original polarity. By the definition of a_V^T and b_V^T on conjunctions, the lemma holds. □

Lemma 6.2 (monotonicity of renaming) *For a vertex V and renamings $\mathbf{R}$ and $\mathbf{R}'$ with $\mathbf{R}' \subseteq \mathbf{R}$,*

$$p_r^s(V, \mathbf{R}) \leq p_r^s(V, \mathbf{R}') \leq p^s(V)$$

Proof This follows from the definitions of p_r^s and p^s: both increase monotonically with tree depth. As renaming effectively prunes part of the tree, replacing it with a smaller tree (a single vertex), it can only reduce the values of the functions. □

Lemma 6.3 (monotonicity of coefficients) *For a vertex V in T and renamings $\mathbf{R}$ and $\mathbf{R}'$ with $\mathbf{R}' \subseteq \mathbf{R}$,*

$$a_V^{\mathrm{ren}(T,\mathbf{R})} \leq a_V^{\mathrm{ren}(T,\mathbf{R}')}$$

$$b_V^{\mathrm{ren}(T,\mathbf{R})} \leq b_V^{\mathrm{ren}(T,\mathbf{R}')}$$

Proof This follows from the definitions of a_V^T and b_V^T and Lemma 6.2 above. The value of both coefficients is determined by its ancestors; an ancestor only has a value greater than 1 if multiplication by p^s occurs; by appeal to Lemma 6.2, we see that these values decrease with increasing renaming sets, and hence the coefficients behave similarly. □

The following lemma is used frequently for decomposing the benefit function in the proofs so is stated here in generality.

Lemma 6.4 (comparison of products and sums) *For $p, q \in \mathbb{N}$, the condition $pq - (p + q) \geq 0$ is equivalent to*

$$(p \geq 2 \wedge q \geq 2) \vee (p = 0 \wedge q = 0)$$

If either or both of p and q is known to be non-zero, then the condition becomes

$$p \geq 2 \wedge q \geq 2$$

Proof This follows trivially from the usual rules of integer arithmetic. □

Notice that the benefit function is given in terms of $p^s(V)$, a_V^T and b_V^T; By definition, $p^s(V) \geq 1$, so the second form given in the lemma applies.

We give two lemmas which allow us to simplify the coefficients and clause-counting functions in certain cases.

Lemma 6.5 (clause counting under restricted renaming) *Given a vertex V and a renaming* **R**,

$$p_r^s(V, \mathbf{R}_{\sqsupset V}) = p^s(V)$$

and for a left-hand child vertex V,

$$p_r^s(V, \mathbf{R}_{\sqsupset sib(V)}) = p^s(V)$$

Proof By the definition of $\mathbf{R}_{\sqsupset V}$, none of the descendant vertices of V are in the considered renaming. □

Lemma 6.6 (definition of coefficients) *For a right-hand child E of vertex V with $sign(E) = +$,*

$$b_E^{\mathrm{ren}(T,\mathbf{R}_{\sqsupset E})} = b_E^{\mathrm{ren}(T,\mathbf{R}_{\sqsupset E})} p_r^-(sib(E), \mathbf{R}_{\sqsubset sib(\mathbf{E})})$$

Proof This follows from the definitions of $\mathbf{R}_{\sqsupset E}$ and p_r^s. The value of $p_r^s(sib(E))$ depends only on the descendent subtree of $sib(E)$ which is given by $\mathbf{R}_{\sqsubset E}$. By definition, this is included in $\mathbf{R}_{\sqsupset E}$ if $E = right(V)$. □

6.3.1 Children of Positive Polarity Conjunctions

Lemma 6.7 (renaming children of conjunctions) *Neither conversion renames the children of positive polarity conjunctions. That is, if we define the set of children vertices of positive conjunctions* **pc** *as*

$$\mathbf{pc} = \{V \in \mathbf{V}_I \mid \mathit{inedge}(V) \neq T \wedge \mathrm{pol}(T, \mathit{source}(\mathit{inedge}(V))) = 1\}$$

then

$$\mathbf{pc} \cap \mathrm{BDLT}(T, T) = \emptyset \quad \textit{and}$$
$$\mathbf{pc} \cap \textsc{Comp}(T, T) = \emptyset$$

Proof The argument for the compact conversion follows trivially from its definition.

For the Boy de la Tour conversion, we focus first on vertex X in Figure 6.7a. The benefit of renaming, $B(T, X)$, is evaluated in the context of the renaming $\mathrm{ren}(T, \mathbf{R}_{\sqsupset X})$. Using the identity from Table 6.2 that $a_X^{\mathrm{ren}(T,\mathbf{R}_{\sqsupset X})} = a_B^{\mathrm{ren}(T,\mathbf{R}_{\sqsupset X})}$, we find that the benefit is given by

$$a_B^{\mathrm{ren}(T,\mathbf{R}_{\sqsupset X})} p_r^+(X, \mathbf{R}_{\sqsupset X}) \; - \; \left(a_B^{\mathrm{ren}(T,\mathbf{R}_{\sqsupset X})} + p_r^+(X, \mathbf{R}_{\sqsupset X})\right)$$

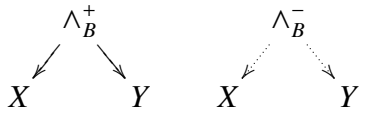
The condition $B(T, X) \geq 0$ therefore reduces to $a_B^{\mathrm{ren}(T,\mathbf{R}_{\sqsupset X})} \geq 2$ and (by Lemma 6.5) $p^+(X) \geq 2$. From Lemma 6.1, in order to satisfy the first condition, vertex B must not be renamed: $B \notin \mathbf{R}$; we deduce from this that $\mathbf{R}_{\sqsupset B} = \mathbf{R}_{\sqsupset X}$. We can now write the condition $B(T, B) < 0$ as

$$a_B^{\mathrm{ren}(T,\mathbf{R}_{\sqsupset X})} p^+(B) \; - \; \left(a_B^{\mathrm{ren}(T,\mathbf{R}_{\sqsupset X})} + p^+(B)\right) < 0$$

Taken together with the other restrictions given above, this leads to the constraint $p^+(B) = 1$.

We now expand $p^+(B)$ to produce a contradiction: since B is a conjunction, $p^+(B) = p^+(X) + p^+(Y) = 1$. However, we observed above that $p^+(X) \geq 2$; since $p^+(Y) \geq 1$ by definition, we deduce that the conditions required to rename X cannot be met.

Following through a similar argument for Y, we find that Y is renamed if $a_B^{\mathrm{ren}(T,\mathbf{R}_{\sqsupset Y})} \geq 2$ and $p^+(Y) \geq 2$ which is satisfied only if B is not renamed. Since $\mathbf{R}_{\sqsupset B} \subseteq \mathbf{R}_{\sqsupset Y}$ we have $a_B^{\mathrm{ren}(T,\mathbf{R}_{\sqsupset B})} \geq 2$ by Lemma 6.3. We can thus again deduce that B is not renamed only if $p^+(X) + p^+(Y) = 1$ and the contradiction follows



(a) Positive (b) Negative

Figure 6.7: RBC subgraphs for the optimality proofs

from $p^+(Y) \geq 2$. Note that we do not need to separately consider whether X was renamed first.

Similar arguments to those given above also apply to the cases where edges BX or BY are signed. □

6.3.2 Children of Negative Polarity Conjunctions

We break the negative polarity argument into several pieces, firstly deriving a simplified version of the Boy de la Tour benefit function.

Consider vertex X in Figure 6.7b. From Table 6.2 we find $a_X^{\text{ren}(T,\mathbf{R}_{\sqsupset X})} = b_B^{\text{ren}(T,\mathbf{R}_{\sqsupset X})} p_r^+(Y, \mathbf{R}_{\sqsupset X})$ which gives us the benefit of renaming $B(T, X)$, in the context $\text{ren}(T, \mathbf{R}_{\sqsupset X})$, as

$$b_B^{\text{ren}(T,\mathbf{R}_{\sqsupset X})} p_r^+(Y, \mathbf{R}_{\sqsupset X}) p_r^+(X, \mathbf{R}_{\sqsupset X}) \; - \; \left(b_B^{\text{ren}(T,\mathbf{R}_{\sqsupset X})} p_r^+(Y, \mathbf{R}_{\sqsupset X}) + p_r^+(X, \mathbf{R}_{\sqsupset X}) \right)$$

which is simplified by Lemma 6.5 to

$$b_B^{\text{ren}(T,\mathbf{R}_{\sqsupset X})} p^+(Y) p^+(X) \; - \; \left(b_B^{\text{ren}(T,\mathbf{R}_{\sqsupset X})} p^+(Y) + p^+(X) \right)$$

There are two cases to consider in order for the condition for renaming, $B(T, X) \geq 0$, to be fulfilled.

1. If $b_B^{\text{ren}(T,\mathbf{R}_{\sqsupset X})} = 1$ then the renaming decision is localised: it is based only on $p^+(X)$ and $p^+(Y)$. In this case we may define a new benefit function $B'(T, X)$:

$$B'(T, X) = p^+(sib(X)) p^+(X) - \left(p^+(sib(X)) + p^+(X) \right)$$

2. If $b_B^{\text{ren}(T,\mathbf{R}_{\sqsupset X})} \geq 2$, we follow the same pattern of reasoning as in the positive polarity case above. By Lemma 6.1, $B \notin \mathbf{R}$ and so $B(T, B) < 0$. This reduces to

$$b_B^{\text{ren}(T,\mathbf{R}_{\sqsupset X})} p^-(B) \; < \; b_B^{\text{ren}(T,\mathbf{R}_{\sqsupset X})} + p^-(B)$$

which is satisfied only by $p^-(B) = 1$. Since B is a negative polarity conjunction, $p^-(B) = p^+(X)p^-(Y)$ and so we deduce that $p^+(X) = 1$. This is sufficient to ensure that $B(T, x) < 0$ and hence X is not renamed. We also notice that in this case, $B'(T, X) < 0$ for $B'(T, X)$ defined in case 1.

We therefore deduce that $B(T, X) \geq 0 \Leftrightarrow B'(T, X) \geq 0$ in either case, and so we can make the same renaming decision in the Boy de la Tour algorithm by adopting the simplified condition $B'(T, X)$ given above. As the condition is the same in both cases, the peripheral knowledge which would allow us to choose between the cases (the value of $b_B^{\text{ren}(T,\mathbf{R}_{\sqsupset X})}$) is also not required.

The above discussion is for vertex X only. The argument for vertex Y in Figure 6.7b is similar, but must take into account the possibility of X being renamed before Y is considered. This prevents one application of Lemma 6.5; we appeal instead to Lemma 6.6 giving the benefit function $B(T, Y)$ in the context $\text{ren}(T, \mathbf{R}_{\sqsupset Y})$ as

$$b_B^{\text{ren}(T,\mathbf{R}_{\sqsupset Y})} p_r^+(X, \mathbf{R}_{\sqsubset X}) p^+(Y) \; - \; \left(b_B^{\text{ren}(T,\mathbf{R}_{\sqsupset Y})} p_r^+(X, \mathbf{R}_{\sqsubset X}) + p^+(Y) \right)$$

Our two cases for $B(T, Y) \geq 0$ are as follows:

1. If $b_B^{\text{ren}(T,\mathbf{R}_{\sqsupset Y})} = 1$ then the renaming decision is based on $p^+(Y)$ and $p_r^-(X, \mathbf{R}_{\sqsubset X})$:

$$B''(T, Y) = p_r^+(sib(Y), \mathbf{R}_{\sqsubset X}) p^+(Y) - \left(p_r^+(sib(Y), \mathbf{R}_{\sqsubset X}) + p^+(Y) \right)$$

2. If $b_B^{\text{ren}(T,\mathbf{R}_{\sqsupset Y})} \geq 2$, we deduce (by Lemma 6.3 as before) that $p_r^+(B, \mathbf{R}_{\sqsupset Y}) = p^+(Y) p_r^+(X, \mathbf{R}_{\sqsubset X})$ and hence that in this case Y is not renamed. As before, we notice that in this case, $B''(T, Y) < 0$, regardless of the value of $\mathbf{R}$.

6.3.3 Both Polarities

Lemma 6.8 (correctness of $B'(T, V)$ and $B''(T, V)$) *For linear trees, the renaming given by the Boy de la Tour algorithm with benefit function $B'(T, V)$ on left-hand vertices and benefit function $B''(T, V)$ on right-hand vertices is the same as with the original function $B(T, V)$.*

Proof The argument for the children vertices of negative polarity vertices is given above (the arguments for different edge signs follow similarly). For

children of positive polarity vertices, it is easy to see that Lemma 6.7 still holds. The remaining case is the root vertex T, which is not renamed under either condition. □

Using this reduced condition, the dependency of the Boy de la Tour on the order of evaluation has been changed, making it more directly comparable with the compact conversion. We define the renaming set construction $\text{BDLT}'(T, V)$ to be a bottom-up construction using the benefit functions $B'(T, V)$ and $B''(T, V)$. From Lemmas 6.7 and 6.8 we know that $\text{BDLT}(T, T) = \text{BDLT}'(T, T)$ for all linear trees T. All remaining theorems are on this bottom-up construction.

Lemma 6.9 (similarity of renaming functions) *For all linear trees T, we have* $\text{Comp}(T, T) \subseteq \text{BDLT}'(T, T)$

Proof We give the proof of the converse (there is no linear tree T such that $\text{Comp}(T, T) \supset \text{BDLT}'(T, T)$) by a *reductio ad absurdam* as it is more convenient. We focus first on vertex X in Figure 6.7a.

Consider the case that $X \notin \text{BDLT}'(T, T)$. From the definition of the Boy de la Tour conversion, $B'(T, X) < 0$. This reduces to two possibilities: either $p(X) = 1$ or $p(Y) = 1$. By Lemma 6.2, choosing a renaming $\mathbf{R}_{\sqsubset B}$, these conditions lead to either $p_r^+(X, \mathbf{R}_{\sqsubset B}) = 1$ or $p_r^+(Y, \mathbf{R}_{\sqsubset B}) = 1$ respectively. In each case, the renaming condition for the compact conversion, given by

$$p_r^+(X, \mathbf{R}_{\sqsubset B}) p_r^+(Y, \mathbf{R}_{\sqsubset B}) > p_r^+(X, \mathbf{R}_{\sqsubset B}) + p_r^+(Y, \mathbf{R}_{\sqsubset B})$$

is hence violated and $X \notin \text{Comp}(T, T)$.

The argument for vertex Y in Figure 6.7a case is similar. Since $B''(T, Y)$ is already defined in terms of $p_r^+(X, \mathbf{R}_{\sqsubset B})$ we only need to apply Lemma 6.2 to one term to deduce that $Y \notin \text{Comp}(T, T)$.

Extended to the other polarities, this means that there is no vertex which is not in $\text{BDLT}(T, T)$ but which is in $\text{Comp}(T, T)$, and the lemma holds. □

The following property about trees before and after renaming is used as part of the subsequent lemma.

Lemma 6.10 (subtree size under renaming) *For all linear trees T, with a renaming $\mathbf{R} = \text{Comp}(T, T)$, for all $V \notin \mathbf{R}$, $p_r^s(V, \mathbf{R}) = 1 \rightarrow p^s(V) = 1$*

Proof We show this by induction on the structure of the tree. The base case $V \in \mathbf{V_L}$ (V is a leaf) is trivial from the definition of p. For the step case, if V is a disjunction, then from the definition of p_r,

$$p_r^s(\mathit{left}(V), \mathbf{R}) = p_r^s(\mathit{right}(V), \mathbf{R}) = 1$$

This means, writing X for $\mathit{target}(\mathit{left}(V))$ and Y for $\mathit{target}(\mathit{right}(V))$,

- $X \notin \mathbf{R}, Y \notin \mathbf{R}$: proof follows from the inductive hypothesis
- $X \notin \mathbf{R}, Y \in \mathbf{R}$: the condition necessary to rename Y,

  $$p_r^s(X, \mathbf{R}_{\sqsubset V}) p_r^s(Y, \mathbf{R}_{\sqsubset V}) \;\leq\; p_r^s(X, \mathbf{R}_{\sqsubset V}) + p_r^s(Y, \mathbf{R}_{\sqsubset V})$$

  is violated because, by Lemma 6.2, $p_r^s(X, \mathbf{R}_{\sqsubset V}) = p^s(X) = 1$.
- $X \in \mathbf{R}, Y \notin \mathbf{R}$: as above, by symmetry
- $X \in \mathbf{R}, Y \in \mathbf{R}$: prohibited by the definition of the compact conversion.

V cannot be a conjunction as $p_r^s(V, \mathbf{R}) \geq 2$ is in contradiction with the induction hypothesis. □

We can now fix the precise difference between the two conversions. Again, we focus first on vertex X in Figure 6.7a. Consider the case $X \notin \textsc{Comp}(T, T)$. By the definition of the compact conversion,

$$p_r^+(X, \mathbf{R}_{\sqsubset B}) p_r^+(Y, \mathbf{R}_{\sqsubset B}) \;\leq\; p_r^+(X, \mathbf{R}_{\sqsubset B}) + p_r^+(Y, \mathbf{R}_{\sqsubset B})$$

which reduces to the three possibilities

- $p_r^+(X, \mathbf{R}_{\sqsubset B}) = 1$
- $p_r^+(Y, \mathbf{R}_{\sqsubset B}) = 1$
- $p_r^+(X, \mathbf{R}_{\sqsubset B}) = p_r^+(Y, \mathbf{R}_{\sqsubset B}) = 2$

In the first case, X may be a leaf vertex, in which case $X \notin \mathrm{BDLT}'(T, T)$, or a disjunction, in which case by Lemma 6.10, $p^+(X) = 1$ and hence[3] $X \notin \mathrm{BDLT}'(T, T)$. A positive polarity conjunction is ruled out by the restriction on the number of clauses. The cases for Y and for signed edges follow similarly. For the final case, by Lemma 6.2, the Boy de la Tour conversion always renames either X or Y: this defines the set of vertices renamed by Boy de la Tour but not by compact.

[3] The case split for BDLT′ is given in the proof of Lemma 6.9

Lemma 6.11 (difference between algorithms) *For all linear trees T,*

$$\textsc{Comp}(T, T) \cup \mathbf{Z} = \text{BDLT}'(T, T)$$

where $\mathbf{Z}$ is the set of vertices such that for all $V \in \mathbf{Z}$,

$$p_r^+(V, \textsc{Comp}(T, V)) = 2 \quad \textit{and} \quad p_r^+(\mathit{sib}(V), \textsc{Comp}(T, V)) = 2$$

Proof From the discussion above and Lemma 6.9, there is no other vertex in $\text{BDLT}'(T, T)$ that is not in $\textsc{Comp}(T, T)$. □

Theorem 6.12 (optimality of compact conversion on linear trees) *The size of the clause form generated by the compact and Boy de la Tour conversions is the same: $p_r^+(T, \textsc{Comp}(T, T)) = p_r^+(T, \textsc{Comp}(T, T))$*

Proof Since renamings may be applied in any order, we show that after applying those in $\textsc{Comp}(T, T)$, the benefit of applying any of those in $\mathbf{Z}$ is zero. By Boy de la Tour's *fundamental theorem of monotonicity* [18], the members of $\mathbf{Z}$ may be considered in any order for this proof.

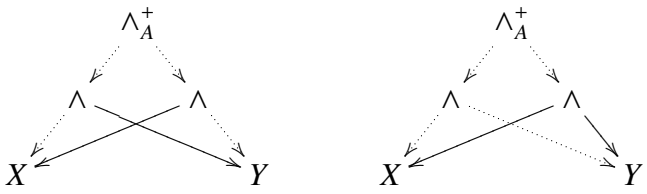
Consider a vertex $X \in \mathbf{Z}$ as depicted in Figure 6.7b. The benefit $B'(T, X)$ of renaming X after $\textsc{Comp}(T, T)$ is given by

$$p_r^+(X, \textsc{Comp}(T, T))p_r^+(Y, \textsc{Comp}(T, T)) - (p_r^+(X, \textsc{Comp}(T, T)) + p_r^+(Y, \textsc{Comp}(T, T)) \quad (6.1)$$

Similarly, consider for a vertex $Y \in \mathbf{Z}$ as depicted in Figure 6.7b. Observing that $p_r^s(V, \mathbf{R}) = p_r^s(V, \mathbf{R}_{\sqsupset V}$, the benefit $B''(T, Y)$ of renaming X after $\textsc{Comp}(T, T)$ is given by

$$p_r^+(Y, \textsc{Comp}(T, T))p_r^+(X, \textsc{Comp}(T, T)) - (p_r^+(Y, \textsc{Comp}(T, T)) + p_r^+(X, \textsc{Comp}(T, T)) \quad (6.2)$$

In each case, by the definition of $\mathbf{Z}$ in Lemma 6.11, and by Lemma 6.2, we deduce that $p_r^+(X, \textsc{Comp}(T, T)) = 2$ and $p_r^+(Y, \textsc{Comp}(T, T)) = 2$, and hence $B'(T, V) = 0$ and $B''(T, V) = 0$. The other polarities follow similarly. □



(a) Positive equivalence (b) Negative equivalence

Figure 6.8: RBC subgraphs for the equivalence discussion

6.3.4 Extension to RBCs

We have shown that the compact conversion produces an optimal number of clauses for linear trees, so we now extend the algorithm to general We now extend the algorithm to RBCs. The extension is heuristic: like Boy de la Tour, we do not claim optimality for the resulting clause form conversion. We first consider equivalences as a special case to achieve an RBC without equivalences but with shared vertices. We then describe how the compact conversion is performed on an conjunction-only RBC with shared vertices.

6.3.4.1 Removal of Equivalences

An RBC with equivalence vertices can be transformed into a linear RBC with only a linear increase in size by replacing equivalences with the subgraphs given in Figures 6.8a and b. These give the minimal linear RBCs representing positive and negative equivalence respectively—equivalences of polarity zero are treated as in the following section. As there is no duplication of vertices we simply see an introduction of two extra conjunctions for each equivalence vertex.

The different treatments for positive and negative polarity equivalences reduce the number of clauses generated, as discussed in Section 2.2.1. Note that a negative equivalence is replaced by a positive subgraph so the incoming edge must have its sign inverted.

6.3.4.2 Polarity Zero Vertices

The children of equivalence nodes are referenced both positively and negatively (as can be seen explicitly from the replacement subgraphs), and hence have zero polarity. Similarly, the sharing used in RBCs encourages a single vertex to be referenced with both polarities. We can convert an RBC with zero polarity

vertices to one without by splitting every zero polarity vertex into a pair, one of each polarity, and suitably treating the incoming edges. Such treatment results in at most a doubling of the size of the RBC.

The substitution and subsequent splitting of equivalences differs significantly from the direct treatment of Boy de la Tour. In particular, Boy de la Tour's algorithm renames a descendant vertex of an equivalence both positively and negatively, simultaneously. This sometimes results in a trade-off: the renaming of one polarity must have sufficient benefit to outweigh any negative benefit of renaming the other polarity. By splitting the polarities and treating them independently we improve the flexibility of the conversion and reduce the number of clauses in some circumstances, as compared to Boy de la Tour.

6.3.4.3 Shared Subgraphs

Having removed equivalences and zero polarity vertices we are close to a linear tree structure. In fact, we can see the resulting structure as a collection of trees joined at the shared vertices. We can incorporate treatment of shared vertices into the bottom-up compact conversion algorithm by renaming any shared vertex which generates more than one clause—the set given by

$$\left\{V \in \mathbf{V} \mid \mathit{inedges}(V) > 1 \wedge p_r^{\mathrm{pol}(T,V)}(V, \textsc{Comp}(T, V))\right\}$$

—and repeating the subgraph otherwise. The resulting algorithm may be considered "locally optimal" as each constituent tree is optimally converted and the shared subgraphs are renamed only when renaming does not increase the resulting size.

6.3.5 Implementation

We have implemented the compact conversion extended to RBCs as part of the NuSMV model checker [22]. The implementation works directly on RBCs, performing the substitutions and duplications described above implicitly rather than constructing the resulting graph explicitly. Each vertex is considered as both a positive and a negative polarity vertex, and a depth-first traversal is used to mark each vertex with the number of incoming edges in each polarity. A second depth-first traversal produces the clause form directly.

Working bottom-up, each vertex is then annotated with the clauses produced positively and negatively after renaming (ie, for vertex V, the value $p_r^s(V, \mathbf{R}) = \textsc{Cnf}(\text{sub}(V, \textsc{Comp}(T, V)))$), the definitional clauses being saved in a global variable (ie, the clauses corresponding to $\textsc{Cnf}(\text{def}(V, \textsc{Comp}(T, V)))$). Whenever a shared vertex is encountered, it is renamed according to the strategy described above. No explicit computation of $p_r^s(V, \mathbf{R})$ is required: it corresponds to the sizes of the sets of clauses which can be determined by a constant time operation.

6.4 Summary

This chapter presents several algorithms for converting propositional logic to CNF. The algorithms are presented as transformations acting on tree RBCs. The restriction of RBCs to trees (RBCs without sharing) simplifies the presentation. The generalisation to RBCs including sharing is described informally.

- The *standard* CNF conversion (non-renaming) is given as a direct transformation from an RBC to a set of clauses, using set operations $\times$ and $\cup$. The standard conversion can result in a worst-case exponential increase in the size of the formula.

- The *structure-preserving* CNF conversion due to Plaisted and Greenbaum [84] is given as a transformation from an RBC to a restructured RBC, which may subsequently be converted to CNF by the standard conversion. The structure-preserving transformation suffers from poor behaviour in certain conditions, especially when the input expression is already in CNF.

- This forms the basis for a general algorithm which renames selected vertices of the RBC according to a *renaming set*—a concept suggested by Boy de la Tour [17]. Three algorithms are given in this form:

  - The structure-preserving transformation is easy to rewrite in this form.

  - The complex but optimal (for trees without $\leftrightarrow$ vertices) Boy de la Tour [17] conversion is given, but it suffers from being hard

to implement efficiently due to the need for arbitrary-size integer arithmetic.

  - A refinement of the Boy de la Tour algorithm given by Nonnengart, Rock, and Weidenbach [81] moves the complexity to a different part of the implementation: a set of elaborate syntactic conditions for determining the vertices to rename.
  - A new CNF conversion (the compact conversion) is given with the advantage of a simple implementation and linear run time.

- The compact conversion is shown to have the same optimality as the Boy de la Tour conversion, and the cases in which the renaming decisions differ in the two algorithms are identified..

Chapter 7

Evaluation of SNF-style Encodings

In this chapter we report the experimental comparison between the BMC encodings presented in the preceding chapters. We consider the standard BMC encoding and three variants of the SNF encoding (identified below). We consider separately the impact of the CNF conversion, since the choice of CNF conversion is independent of the choice of LTL encoding.

7.1 Experimental Framework

We have modified the model checker NuSMV [22] (version 2.1), which includes an implementation of BMC, to include the new encodings presented in Chapter 5 and the compact CNF conversion presented in Chapter 6. We describe briefly the mode of operation of NuSMV and the considerations made when implementing the SNF encoding.

7.1.1 Overview of NuSMV

NuSMV is the unification of standard BDD-based symbolic model checking and the more recent development of bounded model checking. One of the aims of the design has been to share as much code as possible between the two techniques. To this end, the encoding of the model (which we abstracted away in Section 3.2.1) is handled by a two stage conversion, first from the high level input language to a BDD representation, then from the BDD to an RBC (the

underlying propositional logic representation in NuSMV; see Section 2.3.1).

As noted in Section 2.4.2, the model is presented as a symbolic Kripke structure with state variables of integer, Boolean and array types. This is first converted to an *arithmetic decision diagram* (ADD) (a BDD with multiple leaves representing the many possible outcomes) and eventually to a BDD representation of a symbolic Kripke structure defined only in Boolean variables. This Kripke structure has the property assumed in Section 2.4.2, that the atomic propositions correspond to the state variables, and hence the labelling function is the identity function.

NuSMV takes the standard approach to bringing BMC towards completeness by allowing model checking to be performed at a range of different bounds, k_0–$k_{\max}$. By starting at k_0, the fastest performance and the shortest counterexamples are achieved. A maximum bound provides a cutoff point beyond which the extra time spent searching for a counterexample is considered unreasonable. The typical operation flow of NuSMV is as follows:

1. Read model from input file
2. Produce BDD, then RBC, representation of the set of the initial states, $\hat{I}(A)$, and the transition relation, $\hat{T}(A, A')$
3. Initialise the bound k to k_0, the initial bound
4. Create a k-bounded unrolling of the model as an RBC, $R_M = \hat{I}(A^0) \wedge \bigwedge_{0 \leq i < k} \hat{T}(A^i, A^{i+1})$
5. Until $k = k_{\max}$, the maximum bound,
   (a) Read the LTL specification from the input file
   (b) Produce a k-state encoding of the specification
   (c) Call the SAT solver; exit if the specification is satisfied
   (d) Increase the model by an additional transition: $R_M := R_M \wedge \hat{T}(A^k, A^{k+1})$
   (e) Increase k by one

The encoding of the model is extended incrementally with each iteration. Since an RBC representation of $\hat{T}(A, A')$ is constructed initially, a simple replacement of variables in the leaves can be used to construct $\hat{T}(A^k, A^{k+1})$.

The specification, however, is encoded anew with each iteration as the standard encoding cannot be extended in a straightforward way[1]. To test an additional specification against the same model, the RBC unrolling of the model already constructed can be reused. NuSMV is able to extract an appropriate number of states from the completed RBC and hence perform later iterations at a much lower cost.

7.1.2 Implementation

The SNF translation introduces new variables, which become new atomic propositions in the encoding. As noted above, NuSMV is designed to efficiently re-use RBC representations of formulae that have already been constructed. The main implementation challenge was in adding the ability to extend the RBC unrolling of a model with an appropriate number of unconstrained new propositions. This is required because the LTL specification is not known until after the RBC unrolling has been constructed, and so the SNF variables cannot be incorporated into the data structure at the same point as the atomic propositions from the state. The design of NuSMV unfortunately made this a complex operation, as propositions are represented with integer indices, and significant use is made of the assumption that the jth proposition in the ith state can be found at index $ni + j$ for a model with n propositions.

The conversion from LTL to SNF is an implementation of the algorithm given in Section 4.5.3.1, extended with additional optimisation of some common cases using the identities

$$\mathbf{G}\,\phi_0 \wedge \mathbf{G}\,\phi_1 = \mathbf{G}(\phi_0 \wedge \phi_1)$$
$$\mathbf{F}\,\phi_0 \vee \mathbf{F}\,\phi_1 = \mathbf{F}(\phi_0 \vee \phi_1)$$

The optimisation given in Section 5.4.1 is used in all experiments. The SNF conversion is performed when the LTL specification is first read and stored to avoid the overhead of repeated conversions during the iterative deepening search.

[1] This problem was tackled by Benedetti and Bernardini [7].

Table 7.1: Encodings to be evaluated. $|\phi|$ is the number of temporal operators in the LTL specification; f is the number of **F** or **U** operators in nnf($\neg\phi$); k is the size of the bound. Index i indicates the encoding functions enc_c^i, enc_n^i, enc_l^i used

			Asymptotic size		Introduced propositions	
Id	Section	Index	excl. **F**	incl. **F**	excl. **F**	incl. **F**
dfl	3.2		$\|\phi\|k^4$	$\|\phi\|k^4$	0	0
snf	5.3.2.1	2	$\|\phi\|k$	$\|\phi\|k^3$	$\|\phi\|k$	$\|\phi\|k$
lin	5.3.3	4	$\|\phi\|k$	$\|\phi\|k$	$\|\phi\|k$	$\|\phi\|k + f$

7.1.3 Platform

The experiments reported in this chapter were performed on a parallel computing cluster at Edinburgh University. This was used as 64 identical independent 1.8GHz Pentium 4 PCs each with 1Gb of memory running Fedora Core 3.

7.2 Hypotheses

We list the hypotheses of this thesis, as presented informally in Section 1.4.1, in more formal terms.

H0. The size of the clause forms produced using snf is smaller than that produced using dfl.

H1. The time taken by the SAT solver using snf is less than using dfl.

H2. Using lin produces smaller clause forms which are solved faster than with snf.

H3. The reduction in the number of clauses brought about by using the compact CNF conversion in preference to the definitional conversion corresponds to a reduction in the time taken by the SAT solver.

H4. The improvements in solving time do not come at the expense of encoding time.

These hypotheses are tested by the experiments detailed in the following sections.

7.3 Random Experiments

The most significant results in this chapter come from a series of random experiments. The graphs on the following pages show the evaluation of the three encodings (standard BMC, simple SNF, and linear space SNF).

From the hypotheses discussed in Section 7.2 we have several variables to explore. We have designed encodings to reduce the resulting size of the formulae. However, the overall goal of these encodings is to reduce the run time in the SAT solver. The first set of experiments, therefore, is to understand whether the goal of reducing the size of the formulae has been achieved. This is principally a check that the system is implemented correctly, and that the interactions between the parts of the system perform as expected. The second set of experiments, to determine whether run time has been reduced, is less predictable: as noted before, there need not be any correlation between SAT solver run time and formula size.

The parameters to the experiments are the size of the LTL formula and the size of the bound, k.

7.3.1 Models and Formulae

The models and formulae for these experiments are generated using the LBTT [97], a general system designed for comparing LTL to Büchi automaton translations[2]. We have generated a series of 340 problems by varying the number of symbols in the randomly generated LTL formulae from 3 to 17. For each formula size, 20 problems were generated, each with a unique, randomly generated model. These models have 39 states and 6 propositions, and are also generated by LBTT. These problems were checked in NuSMV with the bound k ranging from 1 to 30; where a problem was solvable at a particular k no larger k were tried. This avoids skewing the results at high k in favour of problems which could be quickly solved at low k.

7.3.2 Clauses and Propositions

The results in this section concern the numbers of clauses and propositions produced by each encoding and conversion. We are therefore testing hypothesis

[2] see Chapter 8 for more details of LTL to Büchi automaton translations.

H0, and part of H2. As a side effect, we verify the assumption in H3 that the compact CNF conversion produces fewer clauses than the definitional conversion (this is not surprising as the compact conversion is known to be optimal for linear formulae, while the definitional conversion has known pathological inputs).

7.3.2.1 LTL Formula Size

We first consider the size of the formulae generated by each encoding. In every case, the asymptotic propositional formula sizes are related linearly to the LTL formula size; however, the constant factor is related in different ways to k. In Figures 7.1 and 7.3, the numbers of clauses and propositions are shown for $k = 15$; these can be contrasted with the values at $k = 30$ at Figures 7.2 and 7.4.

The most significant trend in this data is that most of the size of the formula comes from the clause form conversion: for $k = 15$, there is little difference between dfl and snf; however there is a dramatic difference between the encodings with and without the compact conversion. Although at $k = 15$, the encodings are barely distinguishable after the compact conversion, the increased slope of dfl can still be clearly seen in Figure 7.1. The graphs for $k = 30$ make the differences between the encodings clearer—the differences become more exaggerated at higher bounds.

The effect of clause form conversions obscures the other trends, so the graphs are repeated in Figures 7.5 and 7.6 using just the compact conversion. Here we can clearly see the different slopes of the different encodings: the general trends reflect the different linear relationships noted in Table 7.1. We can also see the trade-off made by the lin encoding: in Figure 7.6, this encoding uses slightly more propositions: those involved in obtaining the linear asymptotic behaviour.

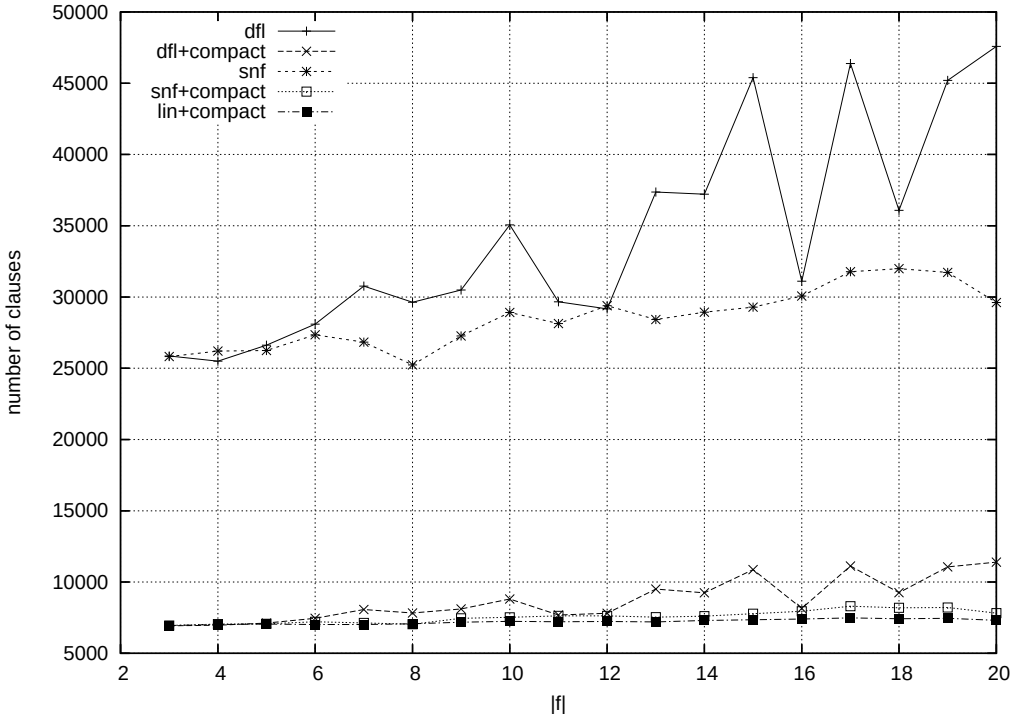


Figure 7.1: $|\phi|$ versus clauses $k = 15$

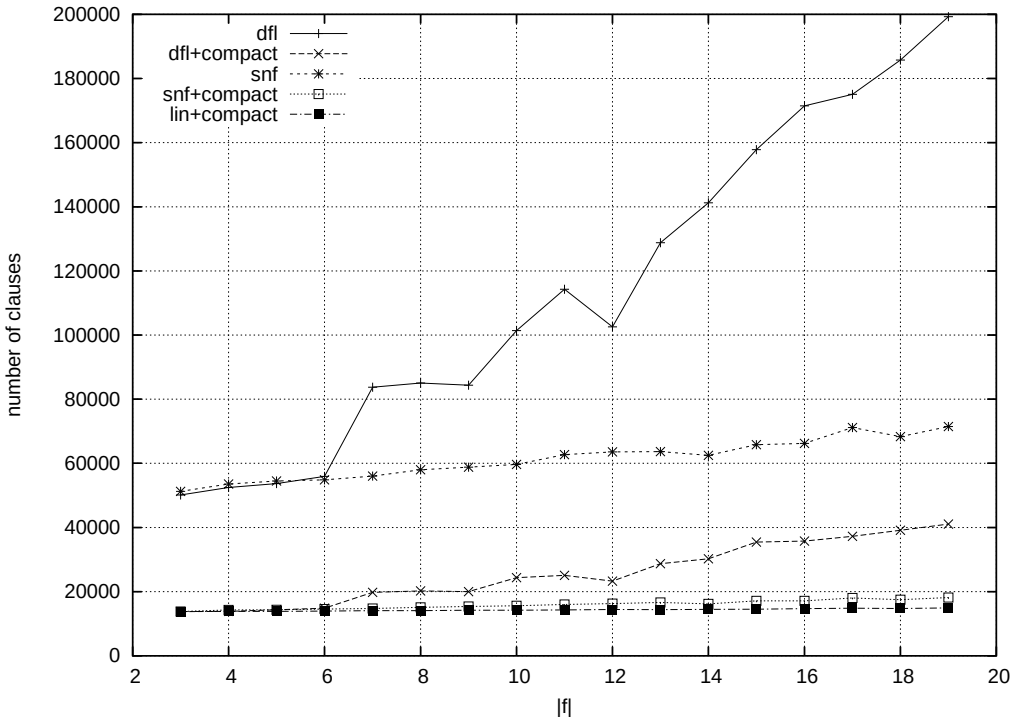


Figure 7.2: $|\phi|$ versus clauses $k = 30$

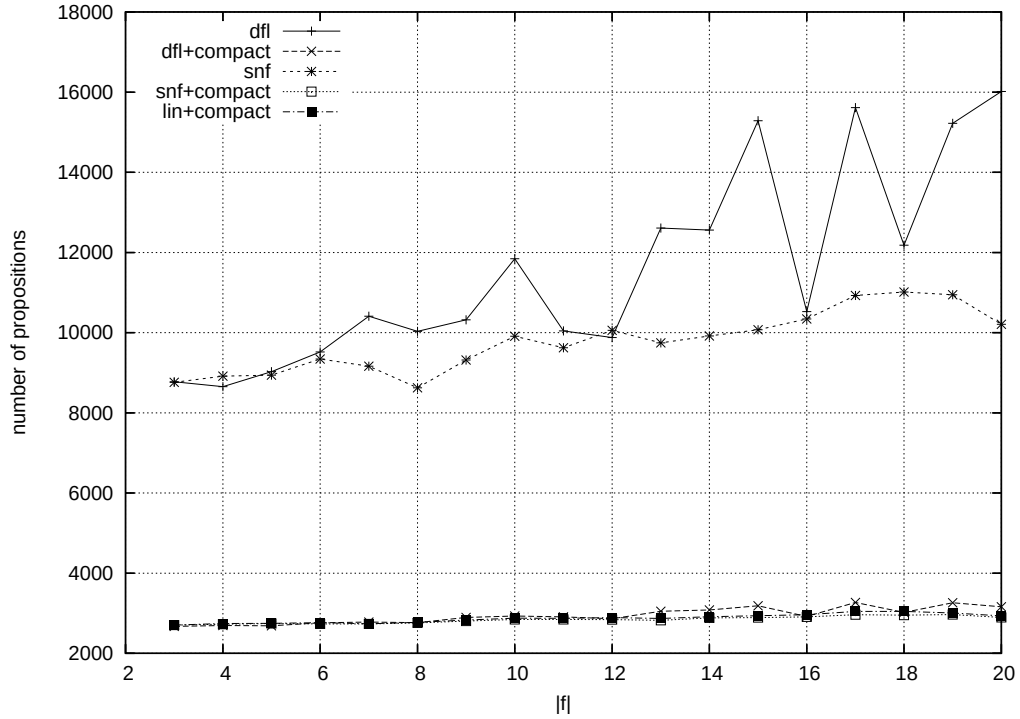


Figure 7.3: $|\phi|$ versus propositions $k = 15$

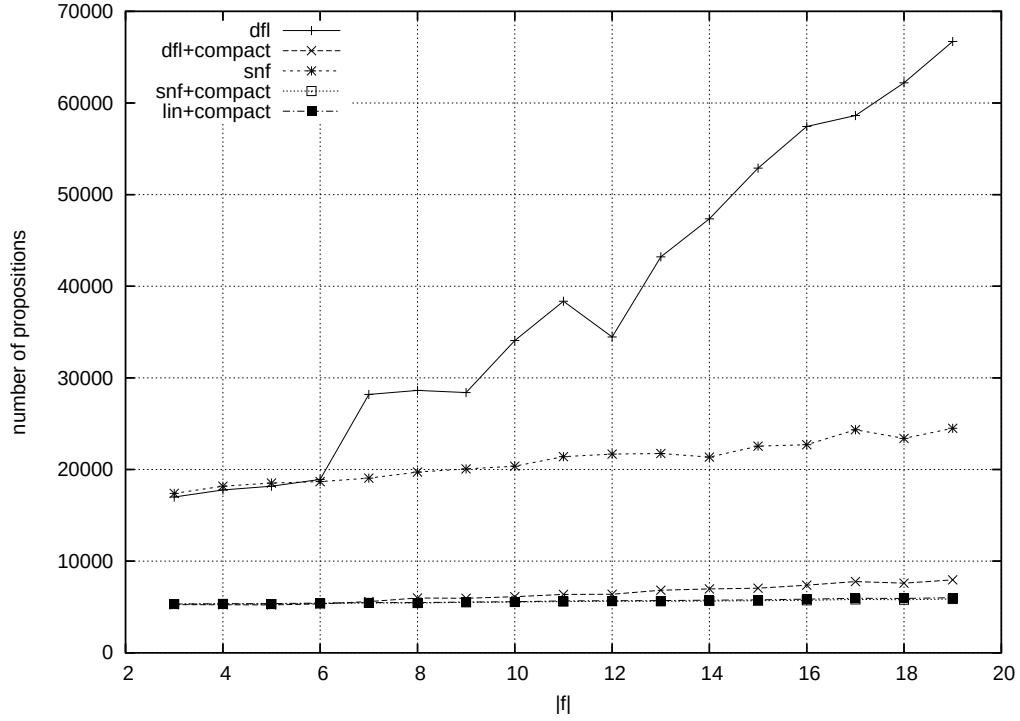


Figure 7.4: $|\phi|$ versus propositions $k = 30$

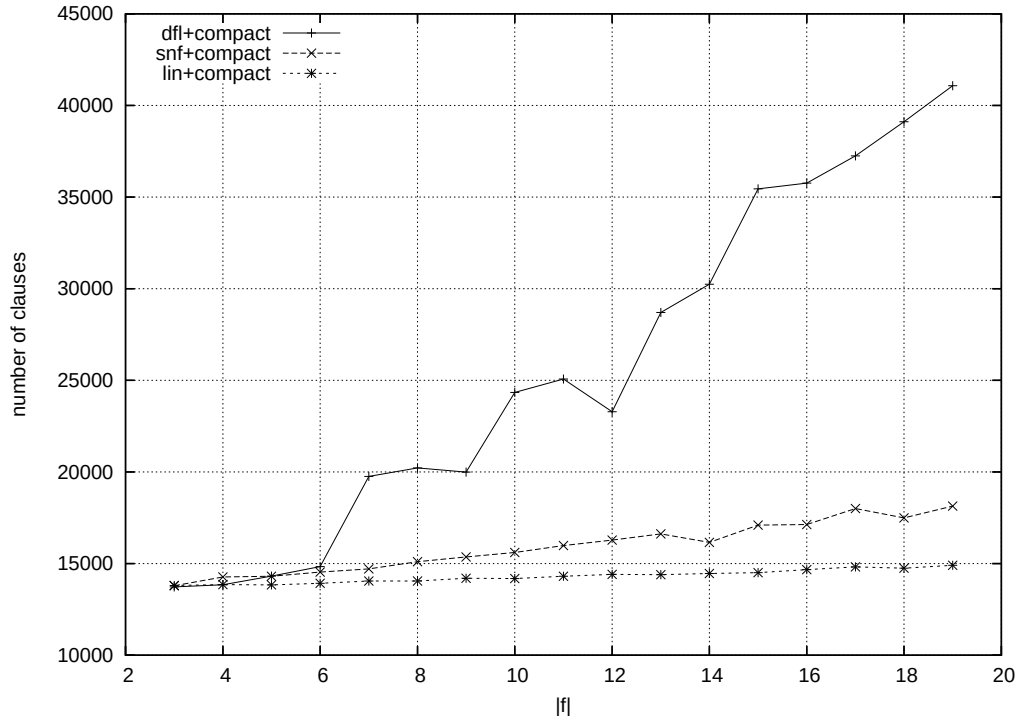


Figure 7.5: $|\phi|$ versus clauses $k = 30$ (compact CNF conversion)

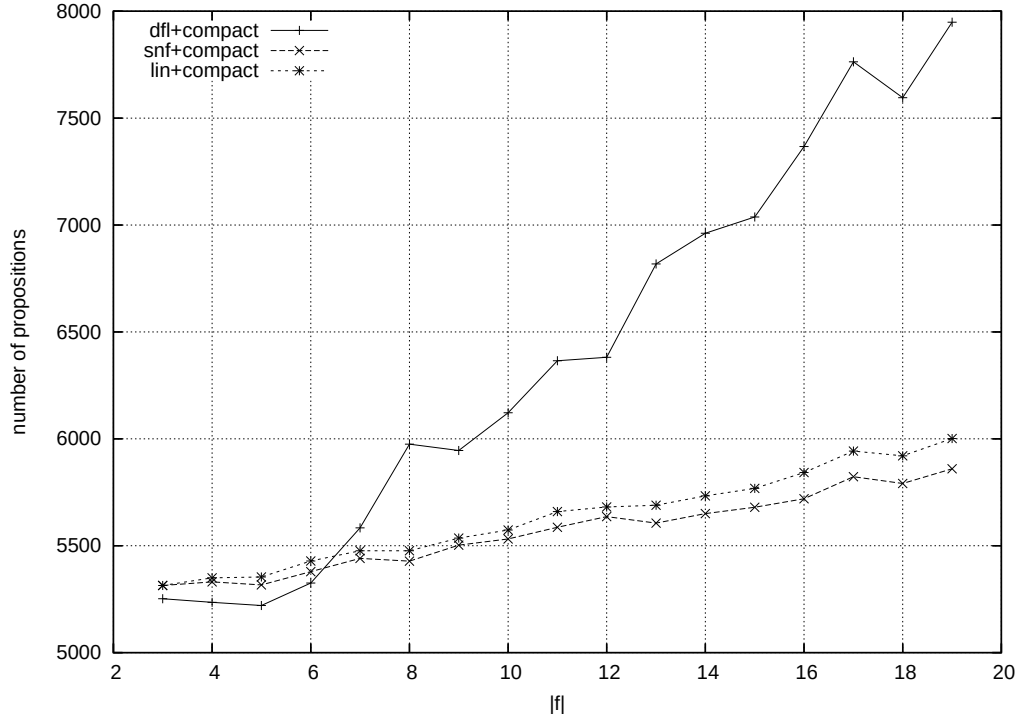


Figure 7.6: $|\phi|$ versus propositions $k = 30$ (compact CNF conversion)

7.3.2.2 Bound

We have already seen the effect of increasing the bound on the various encodings in the previous section. In this section we show the trends in more detail.

The first two graphs (Figures 7.7 and 7.8), for formulae of size three, form a useful control for the experiments. For small LTL expressions, we expect the difference between the encodings to be minimal—smaller LTL formulae mean that the size of the overall propositional expression is dominated by the encoding of the model. As expected, the main impact on size is again the CNF conversion.

A more interesting comparison is given in the following graphs (Figures 7.9 and 7.10), for formula size sixteen. This emphasises the effect of LTL formula size on the relationship. The polynomial growth in the formula size is clearly visible if the outliers are disregarded.

Again, the trends are seen more easily if we focus on the compact CNF conversion only. Figure 7.11 shows the number of clauses. There is little difference between the encodings until k becomes high; for lin the slope is clearly linear as expected. The corresponding graph of the propositions (Figure 7.12) shows no significant difference between the numbers of propositions used by the encodings: perhaps to be expected, as even for large LTL formulae, the encoding of the model is still likely to dominate overall. At the extreme of $k = 30$, however, we see the same trend noted in the previous section: lin requires slightly more propositions than snf. Although dfl does not explicitly introduce new propositions, the larger and more complex propositional formulae mean that more propositions are introduced during CNF conversion.

7.3.2.3 Conclusions

We conclude that this data supports H0: the difference between the encodings grows significantly more marked with larger formulae and the rate of growth of that difference is greater for larger k; the difference is emphasised by the compact conversion. For H2, we see lin produces fewer propositions but more clauses than snf; this hypothesis is only partially supported. The assumption in H3, however, is very clearly supported by the data. In fact, the CNF conversion has a much greater impact on the clause size than the encoding.

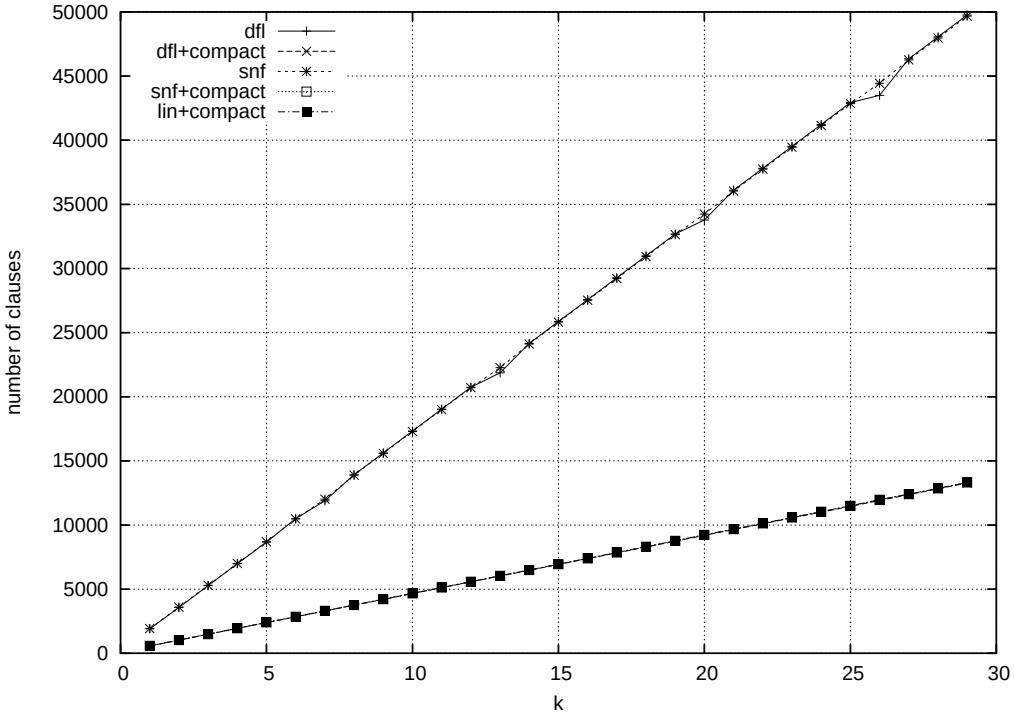


Figure 7.7: k versus clauses $|\phi| = 3$

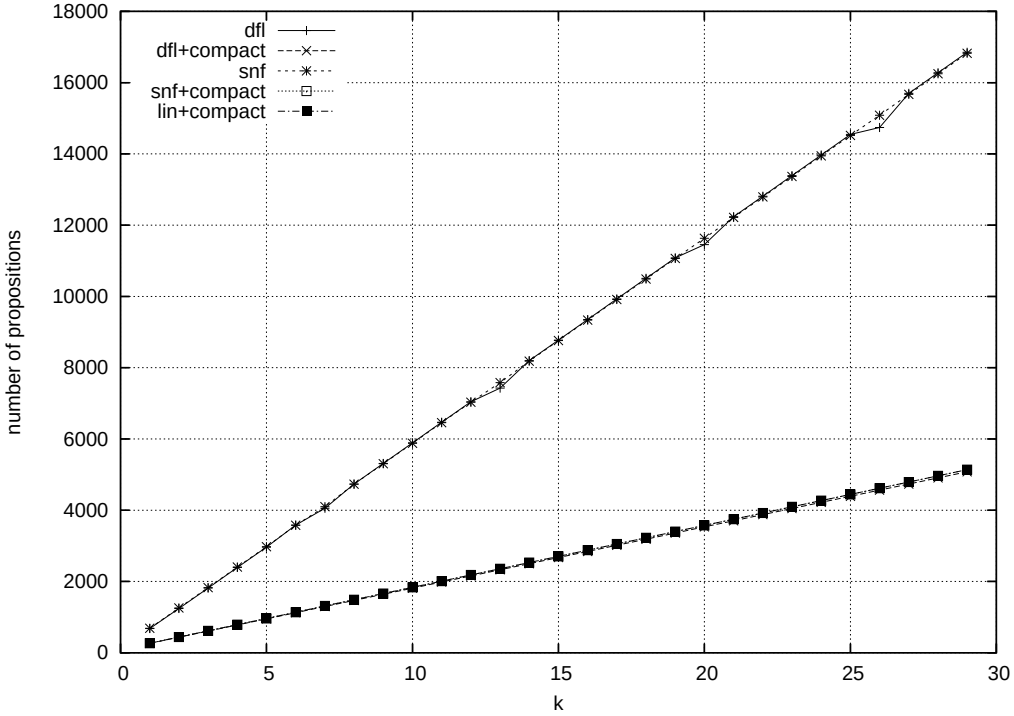


Figure 7.8: k versus propositions $|\phi| = 3$

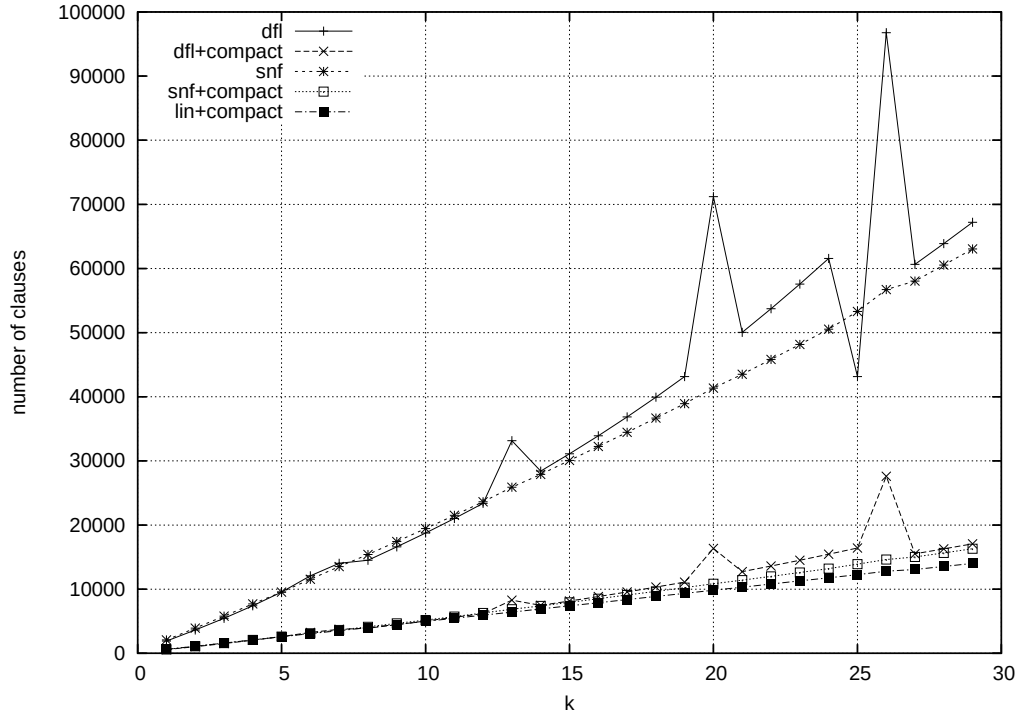


Figure 7.9: k versus clauses $|\phi| = 16$

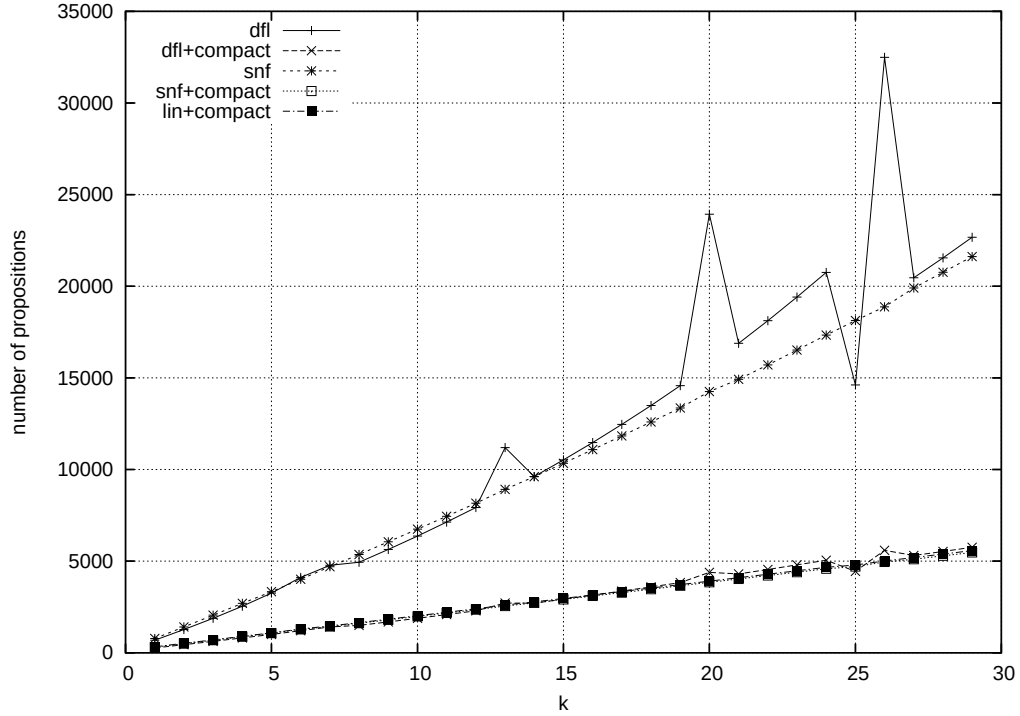


Figure 7.10: k versus propositions $|\phi| = 16$

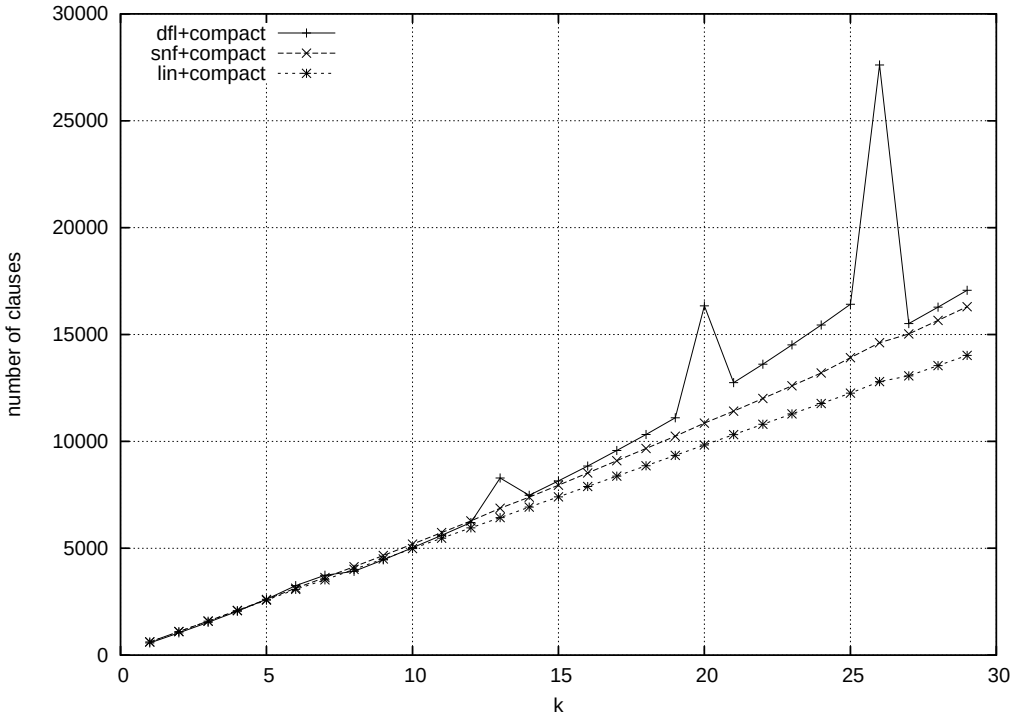


Figure 7.11: k versus clauses $|\phi| = 16$ (compact conversion)

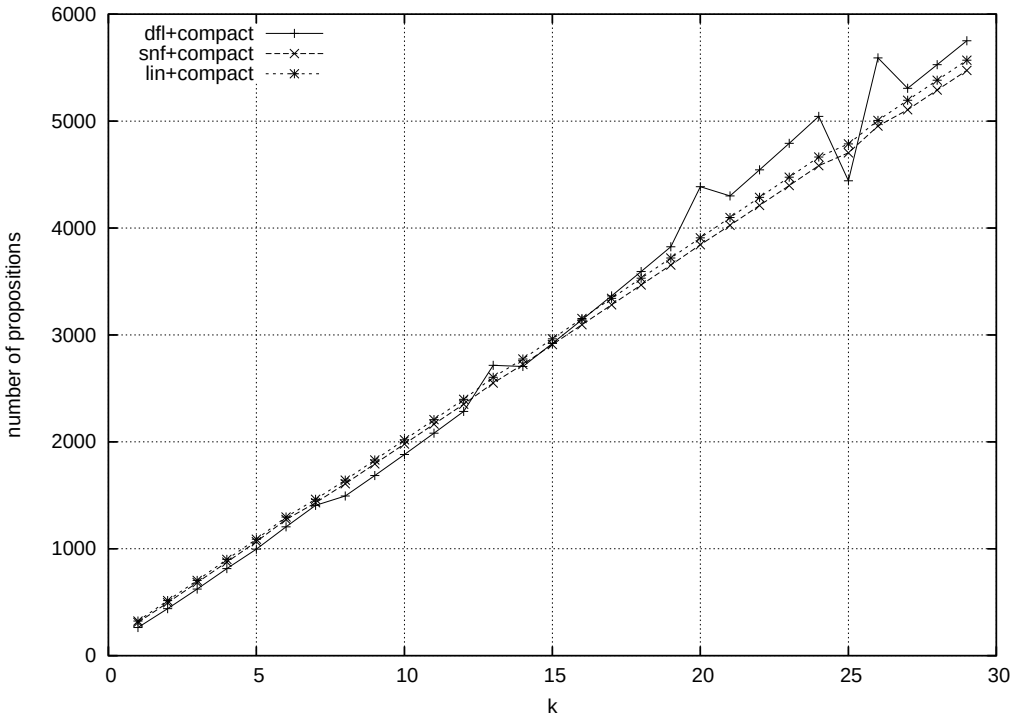


Figure 7.12: k versus propositions $|\phi| = 16$ (compact conversion)

7.3.3 Solving Time

In this section, we investigate the effect of formula size and bound on the time taken to reach a solution, testing hypotheses H1, part of H2, and H3. We investigate the SAT solvers zChaff [79] and BerkMin561 [59] due to their leading positions in the recent SAT competitions [9].

The relationships between solving time and formula size are shown in Figures 7.13 and 7.14. Perhaps the most surprising result is the performance of snf compared to lin: contrary to expectations, snf with the compact CNF conversion is the faster method. For both solvers, the impact of the CNF conversion is much less significant than for the overall numbers of clauses and propositions, although the dfl with the definitional conversion still performs dramatically worse than the other methods.

There is little positive slope in these results, suggesting that the impact of formula complexity is not too great.

Figures 7.15 and 7.16 show the impact on solving time as the bound is increased. Here we can clearly see the polynomial growth of some of the functions, although for small k there is little difference between them. In these graphs, lin appears to beat the other methods most of the time. As before, we have given results for just the compact CNF conversion in Figures 7.17 and 7.18. We see that there are cases where each encoding method is the fastest, but as k increases, lin begins to emerge as the overall winner.

These results support hypothesis H1, with the difference appearing to be more dependent on the bound than the formula size. Hypothesis H2 is partly supported: as k increases, lin shows an advantage, but it is much less clear than for the size results in the previous section. H3 is more clearly supported with all conversions showing a clear, complementary improvement with the compact CNF conversion.

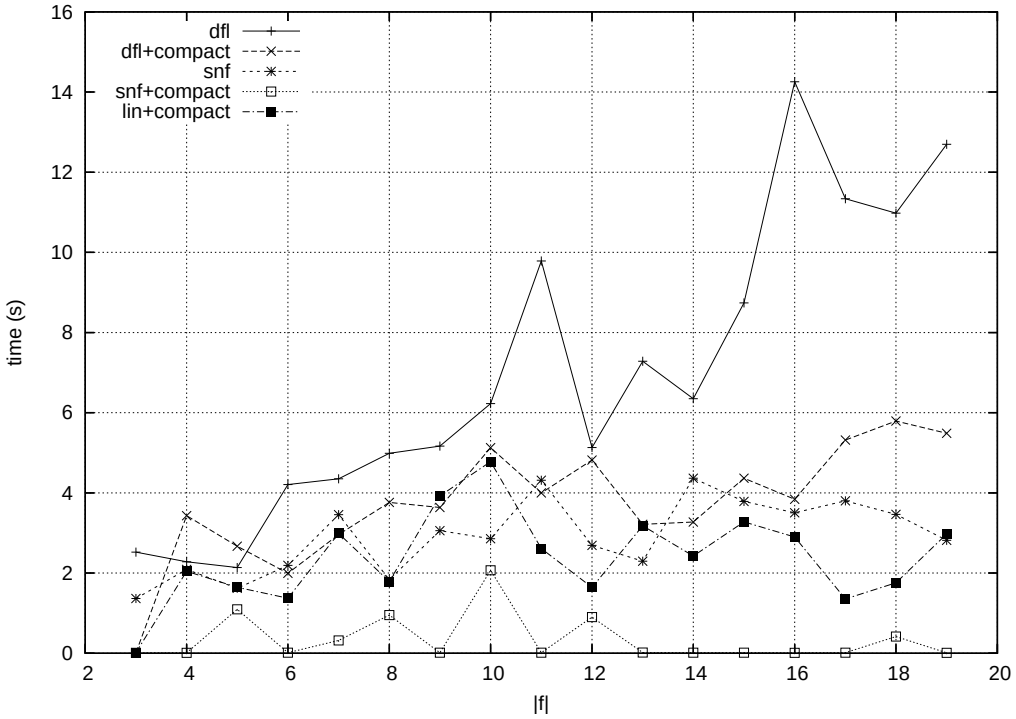


Figure 7.13: $|\phi|$ versus solving time in zChaff at $k = 30$

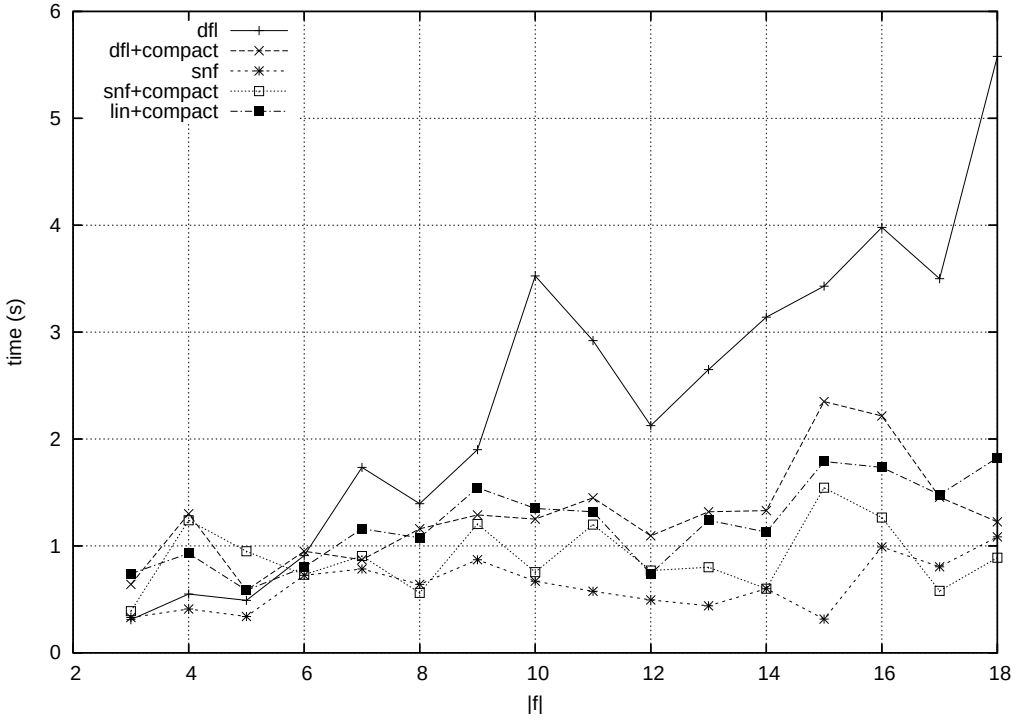


Figure 7.14: $|\phi|$ versus solving time in BerkMin at $k = 30$

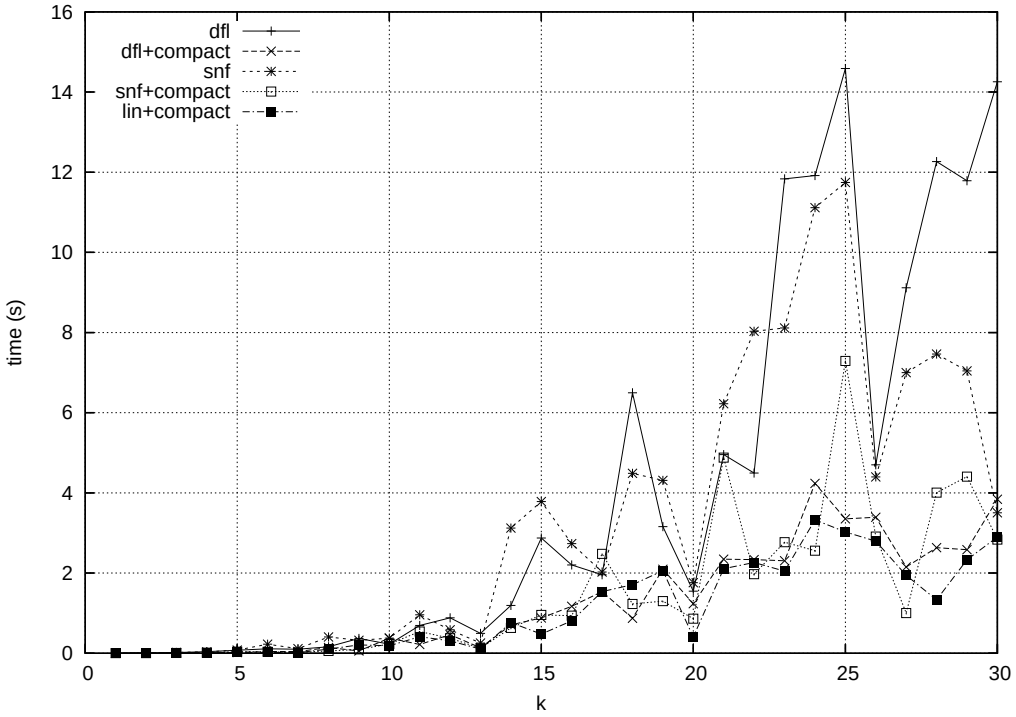


Figure 7.15: k versus solving time in zChaff at $|\phi| = 16$

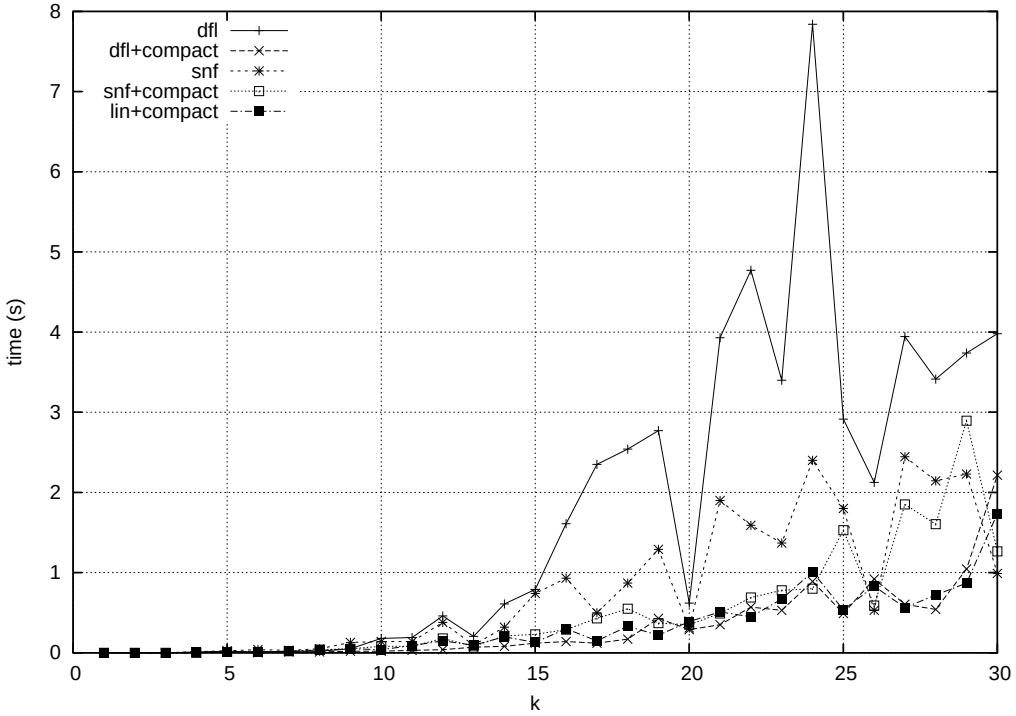


Figure 7.16: k versus solving time in BerkMin at $|\phi| = 16$

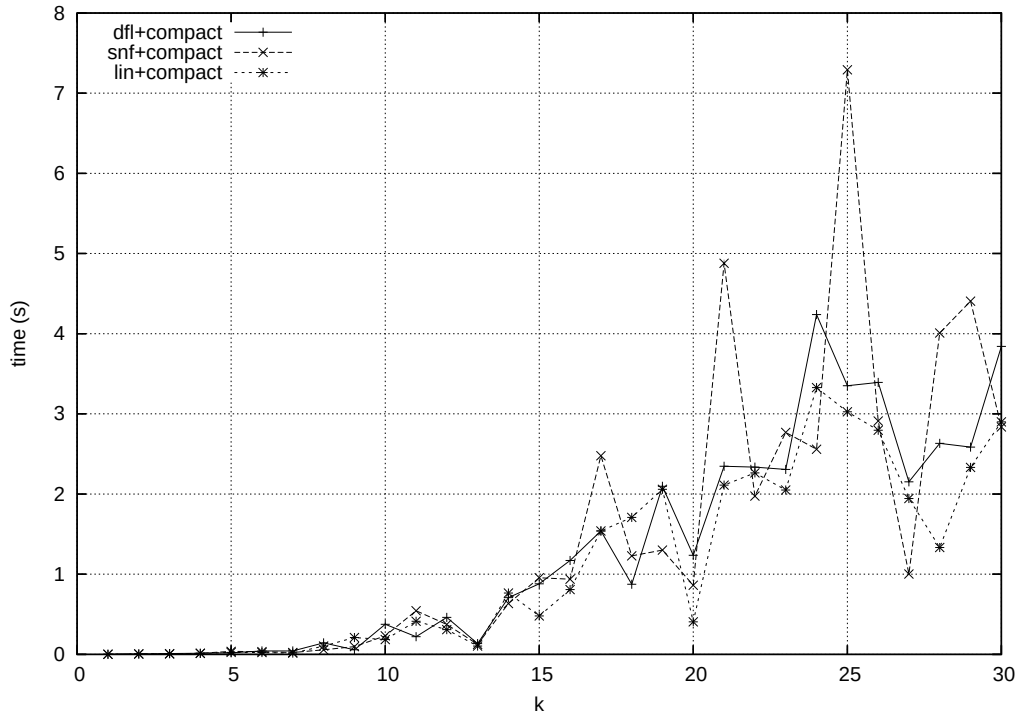


Figure 7.17: k versus solving time in zChaff at $|\phi| = 16$ (compact conversion)

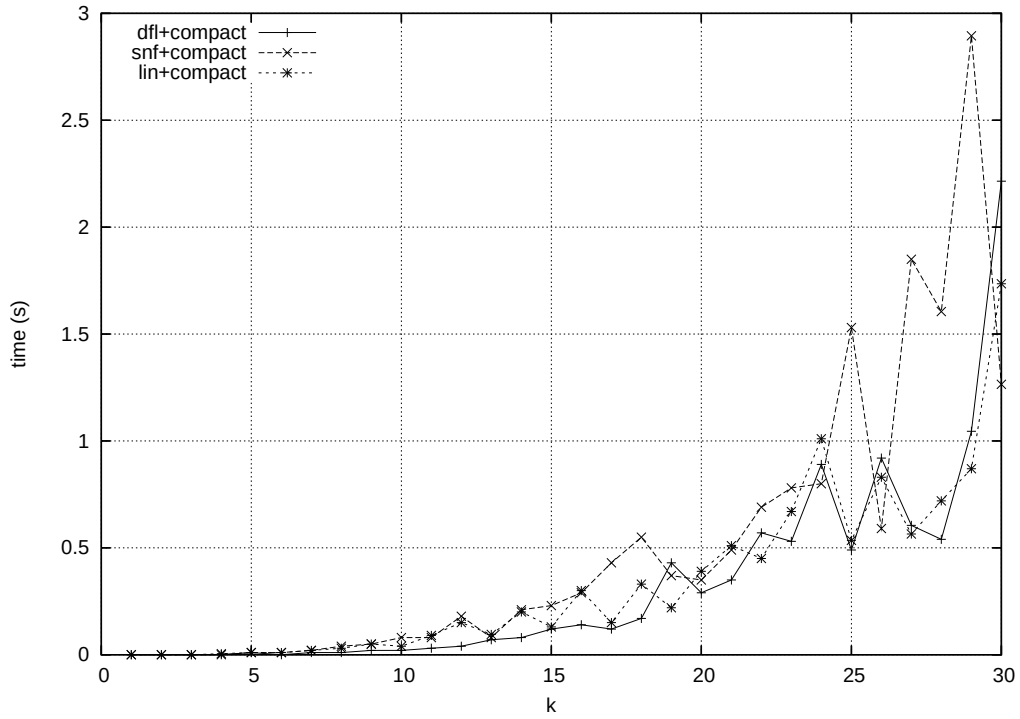


Figure 7.18: k versus solving time in BerkMin at $|\phi| = 16$ (compact conversion)

7.3.4 Encoding Time

Finally, it is important to consider the time taken to produce each encoding: an encoding which resulted in a dramatic decrease in SAT solver time would be useless if it required a dramatic increase in time to create. These experiments test hypothesis H4.

Figures 7.19 and 7.20 show the encoding time versus the LTL formula size and the bound. Interestingly, the CNF conversion again dominates: although the definitional conversion is simpler, the increase in the amount of work required is sufficient to dominate the encoding times. In the second graph, we see that the most advanced conversion, lin, is also the fastest to produce, for larger k. We speculate that this is the result of the simpler logic produced by this encoding; the encoding is closer to clause form than snf due to the extra treatment of eventualities. That is, this graph suggests that the time taken converting to CNF is more significant than the time taken producing the encoding as a whole.

These results therefore clearly support H4: rather than there being a cost in the new encodings and CNF conversion, they also result in reduced encoding times.

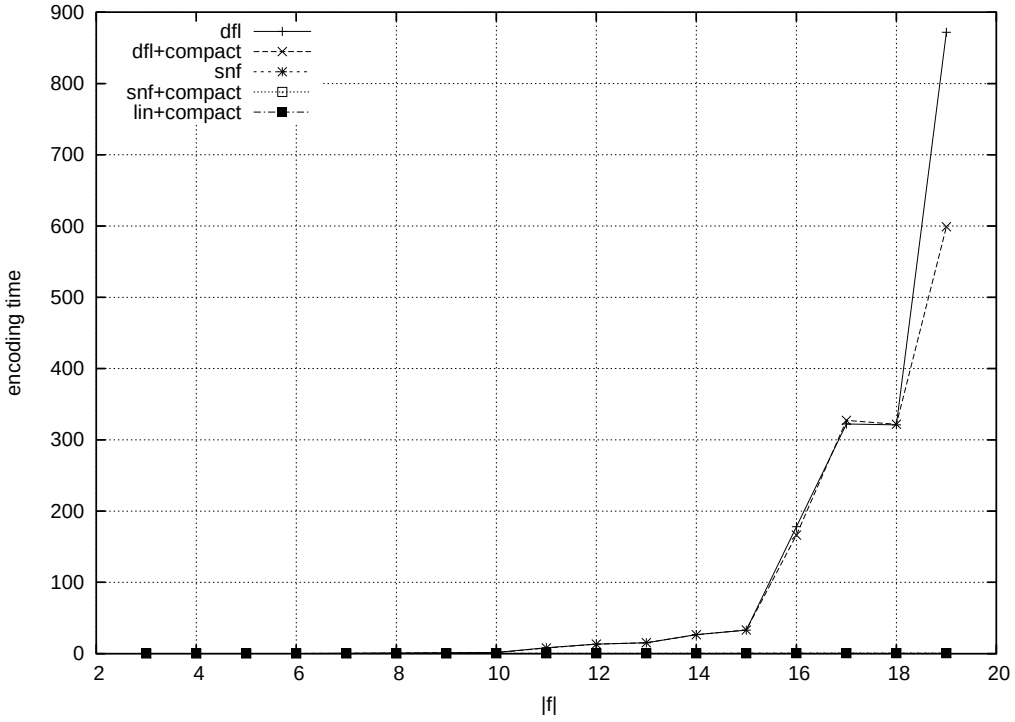


Figure 7.19: $|\phi|$ versus encoding time at $k = 30$

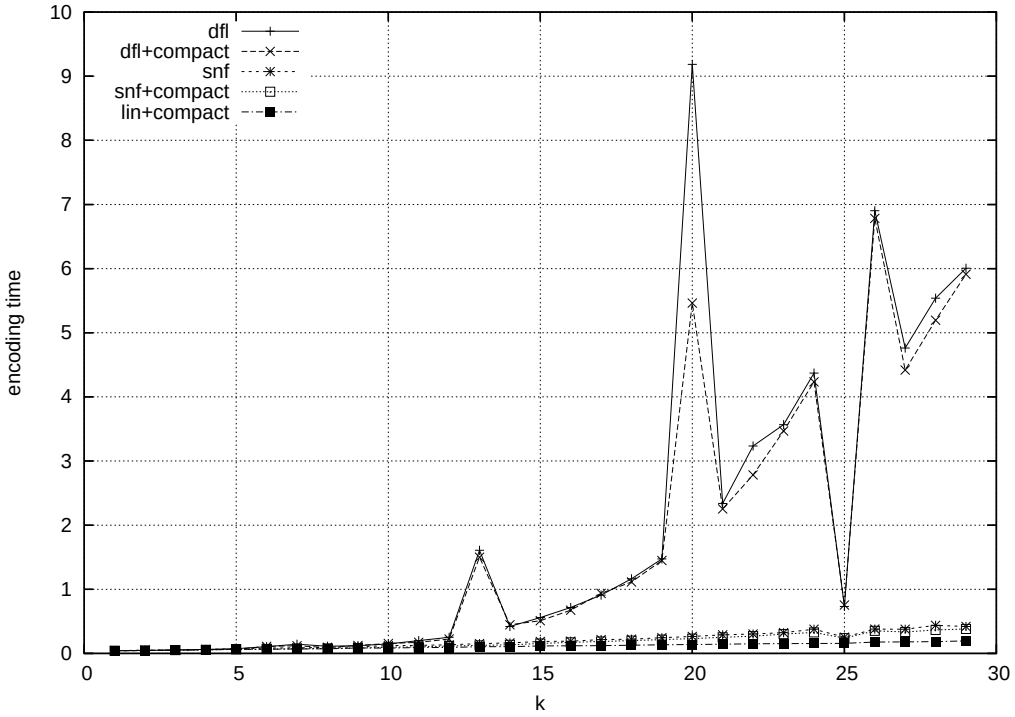


Figure 7.20: k versus encoding time at $|\phi| = 16$

7.4 Distributed Mutual Exclusion

To evaluate the methods' performance on a non-random problem, we use a standard industrial benchmark. Most freely available large benchmarks have very simple specifications: we could not expect any significant difference between the encodings under these circumstances. The distributed mutual exclusion circuit from Martin [74] forms a good basis for comparing the performance of different encodings as it meaningfully implements several specifications. We look at three here, applied to a DME of four elements:

- Accessibility: if an element wishes to enter the critical region, it eventually will. We check the accessibility of the first two elements. This specification is correct, so as in [12], we check at a chosen bound to illustrate the timing differences.

  $\mathbf{G}(\mathit{request}(0) \rightarrow \mathbf{F}\,\mathit{enter}(0)) \wedge \mathbf{G}(\mathit{request}(1) \rightarrow \mathbf{F}\,\mathit{enter}(1))$

- Precedence given token possession: the mutual exclusion property is enforced by a token passing mechanism; if an element of the DME holds the token, then its requests to enter the critical region are given precedence. We check the converse: if the first element holds the token, the second does not have precedence and *vice versa*. Since the token begins at the first element, this is the quicker to prove, with a bound of 14. For the second element, a bound of 54 is required to find the counterexample.

  $\mathbf{G}((\mathit{request}(0) \wedge \mathit{request}(1) \wedge \mathit{token}(0)) \rightarrow (\neg\mathit{enter}(0)\,\mathbf{U}\,\mathit{enter}(1)))$

- Bounded overtaking given token possession: if two elements wish to enter the critical region, then the higher priority may enter a given number of times before the other. We check bounded overtaking of one and two entrances. Both specifications are correct so as above we check at a bound of 40. These specifications are the most complex, including up to four nested *until* operators.

  For one entrance: $\mathbf{G}((\mathit{request}(0) \wedge \mathit{request}(1) \wedge \mathit{token}(0)) \rightarrow ((\neg\mathit{enter}(0) \wedge \neg\mathit{enter}(1))\,\mathbf{U}(\mathit{enter}(0) \wedge \mathbf{X}(\mathit{enter}(0)\,\mathbf{U}((\neg\mathit{enter}(0) \wedge \neg\mathit{enter}(1))\,\mathbf{U}\,\mathit{enter}(1))))))$

  For two entrances: $\mathbf{G}((\mathit{request}(0) \wedge \mathit{request}(1) \wedge \mathit{token}(0)) \rightarrow ((\neg\mathit{enter}(1) \wedge \neg\mathit{enter}(0))\,\mathbf{U}(\mathit{enter}(0) \wedge \mathbf{X}[\mathit{enter}(0)\,\mathbf{U}((\neg\mathit{enter}(1) \wedge \neg\mathit{enter}(0))\,\mathbf{U}(\mathit{enter}(0) \wedge$

$$\mathbf{X}(enter(0)\,\mathbf{U}((\neg enter(1) \wedge \neg enter(0))\,\mathbf{U}\,enter(1)]))))))$$

These specifications are given by Dwyer, Avruning, and Corbett [42]. The results in Table 7.2 are all from zChaff, using the compact clause form conversion. These results reflect the general trends already observed: snf and lin produce smaller clause forms than dfl (H0 is supported); snf is the overall faster method (H1 is supported) as k gets larger, although for small k there is little to choose between the methods (H2 is not supported). We see dfl being let down by its encoding time for the largest specification (H4 is supported). This is reflects NuSMV's exponential-time implementation of the encoding function (NuSMV relies on RBC sharing to achieve the theoretical polynomial size complexity from the straightforward exponential encoding system).

7.5 Summary

This chapter provides comprehensive evidence from random experiments and a standard benchmark to demonstrate that, as the bound and the formula size increase, the newly introduced encodings and clause form conversion reduce the solving time considerably.

The results given suggest that the overhead of achieving a linear space encoding does not bring about significant practical benefits to justify its complexity compared to the quadratic SNF encoding. The remaining hypotheses are all supported, indicating that the stated goals of the encodings (to reduce formula size and solving time) have been achieved; this is without any increase in the encoding time.

Table 7.2: Distributed mutual exclusion benchmark results

Enc.	Bound	Clauses	Vars	Time	Clauses	Vars	Time	Clauses	Vars	Time
		Accessibility			Overtaking 1			Overtaking 2		
dfl	30	15848	2356	0.26	41874	2356	0.37	Encoding > 1800 secs		
	40	21908	3116	1.47	81539	3116	1.92			
	50	28368	3876	9.83	142404	3876	17.69			
snf	30	14298	2418	0.97	14481	2480	0.23	14814	2573	0.25
	40	19038	3198	0.90	19281	3280	0.75	19724	3403	1.08
	50	23778	3978	4.04	24081	4080	2.76	24634	4233	2.22
lin	30	14299	2449	0.72	14483	2511	0.21	14816	2604	0.27
	40	19039	3239	0.89	19283	3321	0.96	19726	3444	0.88
	50	23779	4029	4.43	24083	4131	4.17	24636	4284	2.59

Chapter 8

SNF versus Automata Methods for BMC

One of the primary criticisms[1] of the SNF encoding method for BMC is the apparent similarity it has to the automaton construction used in many other LTL model checking methods (eg, model checking with SPIN [55], LTL model checking with SMV) without making direct use of this extensive body of work. In this chapter we try to address this issue by placing SNF in its correct position in the automata hierarchy, and provide a discussion of the advantages and disadvantages of the approaches.

The backbone of this chapter was presented at the Second International Workshop on Bounded Model Checking [89]: this included the encodings for Büchi and alternating automata (Sections 8.2.4 and 8.3.2) and an informal summary of their advantages and disadvantages (from Section 8.4).

8.1 Background: Automata and LTL Model Checking

Before the introduction of bounded model checking in 1999 [12], LTL model checking was typically carried out by a technique proposed by Burch, Clarke, McMillan, Dill, and Hwang [21]. This involved converting the LTL specification to an automaton expressing the formula and forming the product with the model automaton, using fairness properties to express the Büchi acceptance condition. All fair paths in the product automaton correspond to failure behaviours for the model.

[1] Rejection by the CHARME 2003 reviewers of an earlier version of Cimatti et al. [24].

Research into producing the smallest automaton for a given LTL has been extensive and varied. A large body of literature gives improvements to the "GPVW" conversion algorithm [56] including simplifying the LTL before conversion, and the automaton after conversion (e.g., Etessami and Holzmann [45]) as well as the conversion itself. More recent work by Fritz [49] and Gastin and Oddoux [53] proposes the use of alternating automata as an intermediate representation of the formula. The number of vertices and edges in an alternating automata representation of an LTL formula is linear in the number of operators in the formula. The alternating automaton is therefore a more convenient representation for manipulations and simplifications than a Büchi automaton, since the latter grows exponentially with the number of operators.

The use of LTL to automata conversions as part of bounded model checking was first explicitly suggested by de Moura, Rueß, and Sorea [36]. For past time LTL, Benedetti and Cimatti [8] compare an extension to the direct encoding against a simple GPVW-style conversion. The only published experimental comparison for pure future LTL, given by Clarke, Kroening, Ouaknine, and Strichman [28], is very brief and mainly exercises the LTL simplification available in many automata conversion programs.

Although there are grounds for distinguishing between the direct-to-propositional conversion and the conversions via automata as "syntactic" versus "semantic" [28], we demonstrate here the close correspondence between SNF and alternating automata and their conversion procedures from LTL. We review the use of Büchi automata for BMC and give a new encoding to enable direct use of alternating automata. This allows us to compare more closely the use of the SNF encoding with the use of automata, to explore the advantages and disadvantages of each approach. Finally, we consolidate the discussion with an experimental comparison between the various encodings.

For the following discussion we adopt a similar generalised form of BMC to that given in Section 5.3.2,

$$[\![\hat{M}]\!]_k \wedge \mathrm{enc}_c(\phi, k) \wedge \left(\mathrm{enc}_n(\phi, k) \vee \bigvee_{l=0}^{k-1} \left({}_l[\![\varpi(k) = \varpi(l)]\!]_k \wedge \mathrm{enc}_l(\phi, k, l) \right) \right)$$

where enc_c, enc_n, and enc_l denote the common, finite prefix, and loop encodings. In this chapter we give definitions of these expressions for Büchi and alternating automata conversions.

8.2 Büchi Automata

The first practical conversion algorithm is given in Gerth, Peled, Vardi, and Wolper [56] and is widely referred to as GPVW. The transformation is based on a tableau system.

There have been a large number of papers describing incremental improvements to GPVW. One of the most useful is Daniele, Giunchiglia, and Vardi [33] as it provides a general framework for describing GPVW-like algorithms, together with a methodology for testing.

8.2.1 Overview of GPVW

The original description of GPVW [56] is in two steps. Firstly, a graph is constructed using a tableau-based algorithm. This graph carries the subformulae used during its own construction, but its structure is eventually the same as the Büchi automaton. The second step of the process is to construct acceptance conditions from the extra data and construct the final automaton.

In brief, each graph node carries three pieces of data: the set, *New*, of temporal properties still to be processed; the set, *Old*, of properties that have already been processed; and the set, *Next*, properties that must hold in all successor nodes. A formula in *New* is processed by examining its main connective: conjunctions are split, requiring both conjuncts to be processed individually; disjunctions result in a split of the current node, requiring each disjunct to be processed separately. Literals are transferred to the *Old* after checking for conflicts (if the negation is already in the set, the whole node is discarded). $\mathbf{X}$ is dealt with by putting its argument into *Next*. When all formulae in *New* have been considered, a successor node is created with *New* taking the present node's *Next*.

Temporal properties are dealt with by partial unrollings of their fixpoint characterisations, such as $\varphi \,\mathbf{U}\, \psi \equiv \psi \vee (\varphi \wedge \mathbf{X}(\varphi \,\mathbf{U}\, \psi))$. Rather than create a node with the same *Old* and *Next* fields as one that already exists, a loop is made in the graph — this deals with the self-referential nature of the unrollings.

8.2.2 Improvements to GPVW

8.2.2.1 GPVW+

A series of improvements are given by Gerth, Peled, Vardi, and Wolper [56] with the argument that the original algorithm is simpler to prove—the implication is that the improvements are a necessary in practice. Nevertheless, many authors compare their work firstly with GPVW, referring to the improved algorithm as GPVW+ (we will follow this naming convention).

In summary, the improvements are: $\varphi \mathbf{U} \top \equiv \top$; conflicts in *Old* can be discovered in formulae more complex than literals by conversion to NNF; the set of formulae in *Old* may be simplified with regards conjunction: if φ and ψ are in *Old*, $\varphi \wedge \psi$ is not required; in the case of $\varphi \mathbf{U} \psi$, if ψ is already an obligation of the node, the entire until expression is deemed to be dealt with, and is moved to *Old*.

8.2.2.2 Reducing duplicated cover

Some limited attempts are made in GPVW to avoid covering the same temporal proposition with more than one node by a simple one-to-one matching. More extensive techniques are available: consider, for example, the optimisation concerning removing $\mathbf{U}$ in GPVW+; if a node already exists covering the right argument of the $\mathbf{U}$, the nodes may be merged.

The method used by Daniele, Giunchiglia, and Vardi [33] and Giannakopoulou and Lerda [57] to detect contradictions and redundancies is based on *syntactic implication* defined on a set of formulae A as follows:

- $\top \in \mathcal{SI}(A)$
- $\mu \in \mathcal{SI}(A)$ if $\mu \in A$
- $\varphi \wedge \psi \in \mathcal{SI}(A)$ if $\varphi \in \mathcal{SI}(A)$ and $\psi \in \mathcal{SI}(A)$
- $\varphi \vee \psi \in \mathcal{SI}(A)$ if $\varphi \in \mathcal{SI}(A)$ or $\psi \in \mathcal{SI}(A)$
- $\varphi \mathbf{U} \psi \in \mathcal{SI}(A)$ if $\psi \vee (\varphi \wedge \mathbf{X}(\varphi \mathbf{U} \psi)) \in \mathcal{SI}(A)$
- $\varphi \mathbf{R} \psi \in \mathcal{SI}(A)$ if $\psi \wedge (\varphi \vee \mathbf{X}(\varphi \mathbf{R} \psi)) \in \mathcal{SI}(A)$

A formula φ causes a contradiction if $\neg\varphi \in \mathcal{SI}(\mathit{Old} \cup \mathbf{X}\,\mathit{Next})$. The formula is redundant (that is, it is already covered in some way) if $\varphi \in \mathcal{SI}(\mathit{Old} \cup \mathbf{X}\,\mathit{Next})$

or in the case where $\varphi = \psi \mathbf{U} \mu$ then it is redundant if $\varphi \in \mathcal{SI}(Old \cup \mathbf{X} Next)$ — this special treatment of until is to preserve information that will be needed later on to define acceptance conditions: in the final automaton a formula $\varphi \mathbf{U} \psi$ is accepted at a particular node if $\varphi \mathbf{U} \psi \in \mathcal{SI}(Old \cup \mathbf{X} Next) \rightarrow \psi \in \mathcal{SI}(Old \cup \mathbf{X} Next)$.

8.2.2.3 Labelling transitions

Typically, Büchi automata have labelled nodes. Giannakopoulou and Lerda [57] argues that labelled transitions can lead to more compact automata as nodes can be more easily merged, and the algorithm is modified to reflect this.

Since transitions rather than states are labelled, nodes whose *Next* fields are the same are recorded as being members of an equivalence class. This extra information is used later in the construction of the automata: the nodes during construction will eventually refer to different transitions in the final automaton.

The algorithm notes those formulae which form the eventual obligations of each node: at each node if a formula is seen which was the right hand side of a $\mathbf{U}$ operator, it is added to the set *Eventualities*; any formulae with main connective $\mathbf{U}$ are also added. Thus in the final automaton a formula $\varphi \mathbf{U} \psi$ is accepted at a particular node if $\varphi \mathbf{U} \psi \in Eventualities \rightarrow \psi \in Eventualities$.

8.2.2.4 Symbolic construction

GPVW-like methods explicitly construct an automaton, which is useful for certain model checking algorithms. An alternative is to construct a symbolic description of the automaton, which may be more compact and more appropriate, especially for symbolic model checkers (including BMC).

The original work of this type is by Clarke, Grumberg, and Hamaguchi [26]; the construction given is based on that of Burch et al. [21], but with a focus on producing an SMV model of the automaton. Constructing the automaton for φ begins not with φ in NNF, but by restricting the operators to $\mathbf{X}$ and $\mathbf{U}$. A variable is introduced for each non-literal elementary subformula of φ (in this case, any subformula of the form $\mathbf{X} \psi$, with $\psi \mathbf{U} \mu$ being expanded to $\mathbf{X}(\psi \mathbf{U} \mu)$). The transition relation is defined over the set of states corresponding to the powerset of these variables. The function $sat(\psi)$ returns the set of states which satisfy ψ. Intuitively, for elementary ψ, this is the set of states labelled

with ψ; other functions are built up in the expected way with $sat(\psi \mathbf{U} \mu) = sat(\mu) \cup (sat(\psi) \cap sat(\mathbf{X}(\psi U \mu)))$. The transition relation is defined simply as

$$R(\sigma, \sigma') = \bigwedge_{\mathbf{X}\mu \in el(\varphi)} \sigma \in sat(\mathbf{X}\mu) \Leftrightarrow \sigma' \in sat(\mu)$$

A fairness constraint is introduced for each *until* subformula of φ:

$$\{sat((\psi \mathbf{U} \mu) \rightarrow \mu) \mid \psi \mathbf{U} \mu \text{ occurs in } \varphi\}$$

8.2.2.5 Monotonicity analysis

Schneider [87] comes to the problem from the point of view of relating the hierarchy of temporal logics to that of automata. The basic translation given is the same as Clarke et al. [26] although past time operators are considered, and the translation is expressed very differently. The observation is made that fairness conditions are needed only to support strong operators; if the circumstances can be identified where the weak operator can replace the strong operator then the fairness condition can be ignored. This is discussed as the monotonicity of operators, over the partial order $\varphi \preceq \psi$ iff $\mathbf{G}(\varphi \rightarrow \psi)$. The monotonicity of a subformula ψ in φ reduces to checking whether φ occurs under an even or odd number of negations: the polarity of the subformula. This means that the condition for introducing fairness constraints above becomes

$$\{sat((\psi \mathbf{U} \mu) \rightarrow \mu) \mid \psi \mathbf{U} \mu \text{ occurs positively in } \varphi\}$$

since all negative occurrences become weak until. It is easy to see what this means to an NNF formula: occurrences of **F** and **U** are given fairness constraints, while occurrences of **G** and **R** are not.

8.2.2.6 Finite Intervals of Interest

In order to further reduce the number of fairness constraints, Schneider notes that fairness constraints may be replaced with reachability constraints ($\mathbf{F}\psi$ rather than $\mathbf{G}\mathbf{F}\psi$) provided that only finite prefixes of paths contribute to the truth of the formula: ψ needs only to hold before the end of some finite interval, rather than for all time. This applies only to a subset of LTL, $\mathsf{TL}_{\mathsf{FG}}$.

The construction introduces for each positively appearing **U** an eventuality of the form $\mathbf{F}(((\psi \mathbf{U} \mu) \rightarrow \mu) \wedge \gamma)$ where γ is the conjunction of all such

eventualities for the subformulae ψ and μ. The trade-off here is clear: while the reduction in fairness constraints is attractive, the reachability constraint may grow arbitrarily complex.

In practice, a modified version of the monotonic conversion above is used to rename out all of the subformulae that are not in $\mathsf{TL}_{\mathsf{FG}}$. The final result includes fairness constraints, as well as reachability constraints. This is either transformed to a standard automaton by use of closure theorems, or it is reduced to a combination of an automaton and a CTL specification—the latter having the best performance in the tests given in by Schneider.

8.2.2.7 LTL Simplification

A key technique for improving the performance of LTL to automata conversions is the preprocessing of LTL formulae. In fact, the preprocessing used in Wring [93] and TMP [45] may be considered in competition with the alternating automata phase in LTL2BA [53] and LTL→NBA [49] since these tools do not do any such preprocessing, rather simplifying the alternating automaton.

We present here a summary of the transformations used by Wring and TMP, in the style of Etessami and Holzmann [45]. They are based on closure properties of languages defined by LTL which are obtainable by syntactic means: left-append closedness and suffix closedness. Note that Wring also includes a series of simplifications based on a hierarchy of LTL formulae which we do not cover here.

All pure eventuality formulae (that is NNF LTL with an **F** occurring operator in every branch of the parse tree) are left-append closed, and for a left-append closed formula ϕ, $\psi \,\mathbf{U}\, \phi \equiv \phi$ and $\mathbf{F}\,\phi \equiv \phi$. Similarly, all purely universal formulae (those with **G** in every branch of the parse tree) are suffix closed, and for a suffix closed formula ϕ we have $\psi \,\mathbf{R}\, \phi \equiv \phi$ and $\mathbf{G}\,\phi \equiv \phi$. This characterisation is useful because it allows a large number of simplifications to be covered with easily defined tests. For example, the following properties [93] are subsumed by the above:

$$\mathbf{F}\,\mathbf{G}\,\mathbf{F}\,\phi \equiv \mathbf{G}\,\mathbf{F}\,\phi \qquad \mathbf{G}\,\mathbf{F}\,\mathbf{G}\,\phi \equiv \mathbf{F}\,\mathbf{G}\,\phi$$
$$\mathbf{G}\,\mathbf{G}\,\mathbf{F}\,\phi \equiv \mathbf{G}\,\mathbf{F}\,\phi \qquad \mathbf{F}\,\mathbf{F}\,\mathbf{G}\,\phi \equiv \mathbf{F}\,\mathbf{G}\,\phi$$

The distributivity of **F** over $\wedge$ and of **G** over $\vee$ means that the closure simplifications are also distributable: if ψ is left-append closed then $\mathbf{F}(\phi \wedge \psi) \equiv \mathbf{F}\,\phi \wedge \psi$;

if ψ is suffix closed then $\mathbf{G}(\phi \vee \psi) \equiv \mathbf{G}\,\phi \wedge \psi$. For example,

$$\mathbf{F}(\phi \wedge \mathbf{G}\,\mathbf{F}\,\psi) \equiv (\mathbf{F}\,\phi) \wedge (\mathbf{G}\,\mathbf{F}\,\psi) \qquad \mathbf{G}(\phi \vee \mathbf{F}\,\mathbf{G}\,\psi) \equiv (\mathbf{G}\,\phi) \vee (\mathbf{F}\,\mathbf{G}\,\psi)$$
$$\mathbf{G}(\phi \vee \mathbf{G}\,\mathbf{F}\,\psi) \equiv (\mathbf{G}\,\phi) \vee (\mathbf{G}\,\mathbf{F}\,\psi) \qquad \mathbf{F}(\phi \wedge \mathbf{F}\,\mathbf{G}\,\psi) \equiv (\mathbf{F}\,\phi) \wedge (\mathbf{F}\,\mathbf{G}\,\psi)$$

An additional useful property not identified in the reference is that if ϕ is both suffix and left-append closed, $\mathbf{X}\,\phi \equiv \phi$; the distributivity for $\mathbf{G}$ and $\mathbf{F}$ also applies here. $\mathbf{X}$ may also be pulled through other operators allowing, for example,

$$(\mathbf{X}\,\phi)\,\mathbf{U}(\mathbf{X}\,\psi) \equiv \mathbf{X}(\phi\,\mathbf{U}\,\psi) \qquad (\mathbf{X}\,\phi) \wedge (\mathbf{X}\,\psi) \equiv \mathbf{X}(\phi \wedge \psi)$$
$$\mathbf{X}(\phi \vee \mathbf{G}\,\mathbf{F}\,\psi) \equiv (\mathbf{X}\,\phi) \vee (\mathbf{G}\,\mathbf{F}\,\psi) \qquad \mathbf{X}(\phi \wedge \mathbf{F}\,\mathbf{G}\,\psi) \equiv (\mathbf{X}\,\phi) \wedge (\mathbf{F}\,\mathbf{G}\,\psi)$$

Finally, we note that the following are identified explicitly by both papers

$$(\phi\,\mathbf{U}\,\psi) \vee (\phi\,\mathbf{U}\,r) \equiv \phi\,\mathbf{U}(\psi \vee r) \qquad (\phi\,\mathbf{R}\,\psi) \wedge (\phi\,\mathbf{R}\,r) \equiv \phi\,\mathbf{R}(\psi \wedge r)$$
$$(\phi\,\mathbf{U}\,r) \wedge (\psi\,\mathbf{U}\,r) \equiv (\phi \wedge \psi)\,\mathbf{U}\,r \qquad (\phi\,\mathbf{R}\,r) \vee (\psi\,\mathbf{R}\,r) \equiv (\phi \vee \psi)\,\mathbf{R}\,r$$

8.2.3 Büchi Automata Usage in Practice

Etessami and Holzmann [45] notes that SPIN [55] version 3.3.10 uses GPVW with a combination of some "relatively simple" optimisations in the automata construction with some of the rewrite rules from [45] and [93].

The `ltl2smv` converter which is part of NuSMV is based on the conversion of Clarke et al. [26].

The VIS model checker (The VIS Group[60]) includes implementations of GPVW, GPVW+, LTL2AUT [33], but the default is Wring (Somenzi and Bloem [93]).

8.2.4 Bounded Model Checking with Büchi Automata

In this section we present an encoding for BMC using a Büchi automaton representation of the LTL specification. Just as with the direct and SNF-based encodings already given, we assume that the LTL specification is written in terms of the same set of atomic propositions used to represent the state and labels of the model, $\hat{M} = \langle A, \hat{I}, \hat{T} \rangle$. The Büchi automaton resulting from such an LTL formula therefore has the alphabet $\Sigma = 2^A$. We give a translation from such a Büchi automaton into propositional formulae that constrain these

propositions such that their truth-assignments correspond to the ($k+1$-element-representable) infinite words accepted by the automaton.

The encoding given is a variation on the encoding of de Moura, Rueß, and Sorea [36]. In contrast with this presentation and that of Clarke, Kroening, Ouaknine, and Strichman [28], we use generalised Büchi automata: the complexity of checking multiple acceptance sets is much lower than the overhead of conversion to classical Büchi automata.

By Definition 2.4.2, all paths accepted by a Büchi automaton are infinite. In general, the finite prefix case is therefore never accepting, and we deduce that $\mathrm{enc}_n(\phi, k) = \bot$. We note, however, that a more advanced approach could recognise Büchi automata with finitely representable runs (for example, those derived from $\mathbf{F}\, y$ eventually reach a state with a self-loop which is always taken) and produce an improved encoding in such a case.

Given a generalised Büchi automaton representing LTL formula ϕ, $\mathcal{B}_\phi = \langle Q, \Sigma, \delta, I, \mathcal{T}\rangle$, we define a one-to-one mapping between the states and the first $|Q|$ natural numbers, $\epsilon : Q \to \{0..|Q|-1\}$. We use a new set, $\dot{Q}$, of $\lceil \log_2(|Q|)\rceil$ atomic propositions $q_n \in \dot{Q}$ to represent a state: each possible truth-assignment to the propositions in $\dot{Q}$ corresponds to at most one state.

As in Section 3.2.1, $k+1$ copies of $\dot{Q}$, written $\dot{Q}^i$ for $0 \le i \le k$ are used to represent a run of the automaton. We write $[\![s]\!]^i$ for the property that the ith state in the run is state $s \in Q$. $[\![s]\!]^i$ is a conjunction of propositions $q_n^i \in \dot{Q}^i$ appearing negated or un-negated according to the value of s.

We write the assertion that a set of atomic propositions $\alpha \in \Sigma$, where $\alpha \subseteq A$, holds in state i as $[\![\alpha]\!]^i$, defined as $\bigwedge_{a\in\alpha} a^i$.

We are now able to give the transition relation of the automaton as a set of constraints on pairs of states in the run and on the transition label. If the transition relation is total[2], we can write

$$T_{\mathcal{B}_\phi}(i) = \bigvee_{\langle\hat{\alpha},s'\rangle\in\delta(s)} \bigvee_{\alpha\in\hat{\alpha}} \left([\![s]\!]^i \wedge [\![\alpha]\!]^i \wedge [\![s']\!]^{i+1}\right)$$

[2]The transition relation can be made total for any Büchi automaton. However, we could instead make an assertion that certain state/label combinations are not permitted:

$$T_{\mathcal{B}_\phi}(i,k) = \bigvee_{\langle\hat{\alpha},s'\rangle\in\delta(s)} \bigvee_{\alpha\in\hat{\alpha}} \left([\![s]\!]_k^i \wedge [\![\alpha]\!]_k^i) \to [\![s']\!]_k^{i+1}\right) \wedge \bigwedge_{s\in Q} \bigvee_{\alpha\in\Sigma\setminus\{\alpha|\langle\hat{\alpha},s'\rangle\in\delta(s),\alpha\in\hat{\alpha}\}} \left(([\![s]\!]_k^i \wedge [\![\alpha]\!]_k^i) \to \bot\right)$$

The initial set is encoded directly as a disjunction over members of I:

$$I_{\mathcal{B}_\phi} = \bigvee_{s \in I} [\![s]\!]^0$$

Finally, we encode the acceptance condition for the run. The Büchi acceptance condition is that each member of $\mathcal{T}$ is visited infinitely often. As we have ruled out finite path prefixes, we know that all paths being considered are of the form ab^ω. If we assert as part of the loop encoding that the corresponding paths in the Büchi automaton follow the same pattern, we can simply require that representatives from each acceptance set appear in the loop (ie, in component b):

$$F_{\mathcal{B}_\phi}(k, l) = \bigwedge_{T \in \mathcal{T}} \bigvee_{i=l}^{k} \bigvee_{s \in T} [\![s]\!]^i$$

We give the encoding of Büchi automata for BMC below in terms of constraints on the states in general ($\text{enc}_c(\phi, k)$), constraints on the path resulting from interpreting the states as a finite path prefix ($\text{enc}_n(\phi, k)$), and constraints on the path resulting from interpreting the states as a k-l-loop path ($\text{enc}_l(\phi, k, l)$):

$$\text{enc}_c^{\mathcal{B}}(\phi, k) = I_{\mathcal{B}_\phi} \wedge \bigwedge_{i=0}^{k-1} T_{\mathcal{B}_\phi}(i)$$

$$\text{enc}_n^{\mathcal{B}}(\phi, k) = \bot$$

$$\text{enc}_l^{\mathcal{B}}(\phi, k, l) = F_{\mathcal{B}_\phi}(k, l) \wedge \bigwedge_{i=0}^{\lceil \log_2(|Q|) \rceil - 1} q_i(l) \leftrightarrow q_i(k)$$

Although the LTL to Büchi automaton conversion produces an exponential number of states in the size of the formula, the number of propositions introduced by the encoding above is only linear in the size of the formula. Each component of the resulting encoding produces $O(|\mathcal{T}|k)$ symbols except for $F_{\mathcal{B}_\phi}$ which is quadratic: $O(|\mathcal{T}|k^2)$.

The encoding given above differs from that of de Moura et al. [36] by giving a mapping from the states to the integers, represented in the propositional formula in base-2. de Moura et al. instead use an atomic proposition for each state; the truth of a proposition indicates whether the system is in the state in question. This approach uses exponentially more propositions than that given above, and also requires additional *at-most-one* and *at-least-one* constraints to ensure that the formulae are satisfied only when the truth assignment corresponds to exactly one state.

8.3 Alternating Automata

Alternating automata are a type of tree automaton (runs are described as trees rather than linear traces) combining both deterministic and non-deterministic behaviours: a transition in a non-deterministic automaton leads to a set of states from which one is chosen; a transition in a deterministic tree automaton leads to a successor set, or a conjunction of target states. Alternating automata exhibit the combination of these existential and universal behaviours. Although the presentation that we adopt below is one of a non-deterministic choice between conjunctions of states, it can be generalised to arbitrary propositional formulae over $\wedge, \vee$ and states. Alternating automata are exponentially more succinct than Büchi automata.

There are two presentations of LTL to automata conversion via alternating automata. We follow the slightly unconventional presentation by Gastin and Oddoux [53]: transitions are from a state to a conjunction of states; each state may have multiple transitions and the transition taken is selected non-deterministically. This effectively encodes a disjunction of conjunctions of states reached from a given state.

The game-theoretic presentation given by Fritz [49] is equivalent, but the differences in the definitions lead to larger representations of the automata.

An additional difference is that Gastin and Oddoux use a co-Büchi accepting condition, while Fritz uses a Büchi condition. We can disregard this: for the special case of the alternating automata constructed here, a Büchi condition $F \subseteq Q$ is equivalent to the co-Büchi condition $Q \setminus F$.

Definition 8.3.1 (alternating co-Büchi automaton)
An *alternating co-Büchi automaton* $\mathcal{A}$ is defined by a tuple $\langle Q, \Sigma, \delta, I, F \rangle$ where Q is the set of states; Σ is the alphabet of transition labels; δ is the transition function $Q \to 2^{2^\Sigma \times 2^Q}$; $I \subseteq 2^Q$ is the set of initial combinations of states; $F \subseteq Q$ is the set of final states.

As for the Büchi automaton definition above, the transition labels are from 2^Σ; accepted words are nevertheless from Σ^ω.

Alternating automata representing LTL formulae are known to be *very weak*, which means that there exists a partial order on the states $\langle Q, \sqsubseteq \rangle$ determined by the transitions such that

$$\forall q \in Q, \forall \langle \alpha, s' \rangle \in \delta(q), \forall q' \in s' \;.\; q' \sqsubseteq q$$

That is, transitions are only occur from a state to a state lower or equal in the ordering. The result of this restriction is that the only loops in very weak co-Büchi alternating automata (VWAA) are self-loops.

Definition 8.3.2 (AA run)

A run σ of an alternating co-Büchi automaton on a word $u_0 u_1 \ldots \in \Sigma^\omega$ is a labelled DAG $\langle V, E, \lambda \rangle$ with V partitioned into levels V_i, $V = \bigcup_{i \in \mathbb{N}} V_i$ and $E \subseteq \bigcup_{i \in \mathbb{N}} V_i \times V_{i+1}$. $\lambda : V \to Q$ labels the vertices of the graph with states of the automaton. V_i may be seen as a multiset of elements of Q. The graph is related to the word and the automaton by

$$\lambda(V_0) \in I$$

and

$$\forall v \in V_i, \exists \langle \alpha, s' \rangle \in \delta(\lambda(v)) \;.\; u_i \in \alpha \wedge s' = \lambda(E(v))$$

A run is accepting if every infinite branch of σ has only a finite number of vertices with labels in F.

8.3.1 LTL to VWAA Conversion

We report here the conversion procedure given by Gastin and Oddoux. The set operator $\otimes$ constructs the conjunctions of two sets of disjunctive normal form transitions: $X \otimes Y = \{\langle \alpha_1 \cap \alpha_2, s_1 \cup s_2 \rangle \mid \langle \alpha_1, s_1 \rangle \in X, \langle \alpha_2, s_2 \rangle \in Y\}$. The overbar operator $\bar{\psi}$ converts ψ to a set-style disjunctive normal form representation: a set of conjunctions of atomic propositions, temporal subformulae, their negations, or $\top$.

For an LTL formula ϕ over atomic propositions AP, the corresponding VWAA $\mathcal{A}_\phi = \langle Q, \Sigma, \delta, I, F \rangle$ is given as follows:

- Q is the set of temporal subformulae of Q (the set of subformulae with an LTL operator as the main connective, union the set of atomic propositions and their negations, and $\top$).
- $\Sigma = 2^{AP}$; $I = \bar{\phi}$; F is the set of formulae of the form $\psi_1 \,\mathbf{U}\, \psi_2$ or $\mathbf{F}\, \psi_1$.
- δ is defined as

$$\delta(\top) = \{\langle \Sigma, \top \rangle\}$$
$$\delta(p) = \{\langle \{a \in \Sigma \mid p \in a\}, \top \rangle\}$$

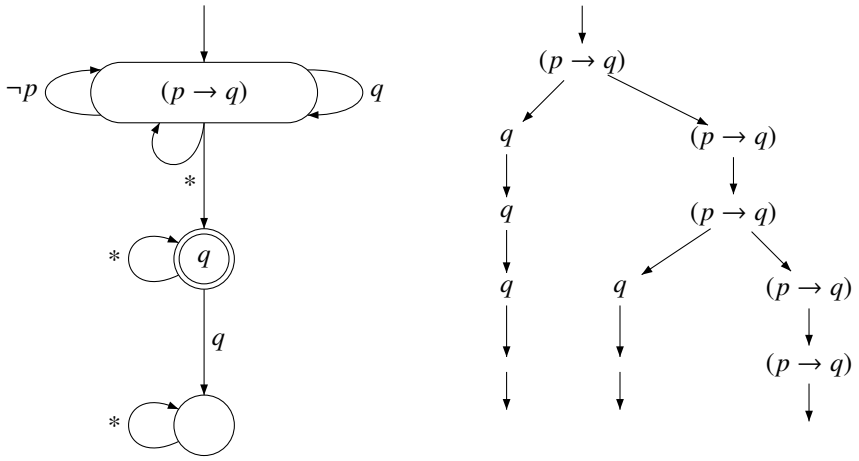


Figure 8.1: Example alternating automaton (left) and part of a run over the input $\sigma = \{p\}\{\}\{p\}\{pq\}\cdots$ (right). '*' indicates the unconstrained transition

$$
\begin{aligned}
\delta(\neg p) &= \{\langle \{a \in \Sigma \mid p \notin a\}, \top\rangle\} \\
\delta(\mathbf{X}\,\psi) &= \{\langle \Sigma, e\rangle \mid e \in \bar{\psi}\} \\
\delta(\mathbf{F}\,\psi) &= \Delta(\psi) \cup (\{\langle \Sigma, \mathbf{F}\,\psi\rangle\}) \\
\delta(\mathbf{G}\,\psi) &= \Delta(\psi) \otimes \{\langle \Sigma, \mathbf{G}\,\psi\rangle\}) \\
\delta(\psi_1\,\mathbf{U}\,\psi_2) &= \Delta(\psi_2) \cup (\Delta(\psi_1) \otimes \{\langle \Sigma, \psi_1\,\mathbf{U}\,\psi_2\rangle\}) \\
\delta(\psi_1\,\mathbf{R}\,\psi_2) &= \Delta(\psi_2) \otimes (\Delta(\psi_1) \cup \{\langle \Sigma, \psi_1\,\mathbf{R}\,\psi_2\rangle\})
\end{aligned}
$$

where Δ is the extension of δ to include the propositional subformulae of ϕ:

$$
\begin{aligned}
\Delta(\psi) &= \delta(\psi) && \text{if } \psi \in Q \\
\Delta(\psi_1 \wedge \psi_2) &= \Delta(\psi_1) \otimes \Delta(\psi_2) \\
\Delta(\psi_1 \vee \psi_2) &= \Delta(\psi_1) \cup \Delta(\psi_2)
\end{aligned}
$$

We give an example VWAA corresponding to the LTL formula $\mathbf{G}(p \rightarrow \mathbf{F}\,q)$ in Figure 8.1 along with a sample run.

8.3.1.1 Compact Representation of Runs

The representation of a run of a VWAA as a DAG is problematic as the number of vertices at each level grows without bound. We can reduce the representation of a run by identifying vertices on the same level that have the same labels

(each level becomes a set rather than a multiset), hence forming a reduced DAG. We call successive sets *configurations*, $C_i \subseteq Q$.

Definition 8.3.3 (reduced AA run)
A reduced run ς of an AA on a word $u_0 u_1 \ldots \in \Sigma^\omega$ is a DAG $\langle C, E \rangle$ where C is a sequence $C_0 C_1 \ldots$ of subsets of Q and $E \subseteq \bigcup_{i \in \mathbb{N}} C_i \times C_{i+1}$. The graph is related to the word and the automaton by

$$C_0 \in I$$

and

$$\forall s \in C_i, \exists \langle \alpha, s' \rangle \in \delta(s) \;.\; u_i \in \alpha \wedge s' = E(s)$$

A run is accepting if every infinite branch of ς has only a finite number of vertices with labels in F.

Alternating automata runs are more expressive than reduced runs, but we can show that if a full (non-reduced) run exists, then a reduced run must also exist; on the other hand, every reduced run is trivially converted to a full run.

Lemma 8.1 (reducability of runs) *An accepting run of AA $\sigma = \langle V, E, \lambda \rangle$ can be converted into a smaller accepting run if two vertices on the same level $v, v' \in V_i$ represent the same state, $\lambda(v) = \lambda(v')$, but have different evolutions $\lambda(E(v)) \neq \lambda(E(v'))$. The reduced run $\sigma' = \langle V, E', \lambda \rangle$ where $E' = E \oplus (v' \mapsto E(v))$.*

Proof If two different transitions are taken from two instances of a given state at a given point in the run, then both must accept the symbols at the given point in the word. The construction above simplifies the DAG by choosing only one transition to take. Since the evolution of the AA from v onwards must have been consistent with the definition of a run, the replacement of the evolution from v' onwards is also consistent with the definition of a run. Since σ is accepting, no branch of σ' has an infinite number of vertices with labels in F, so σ' is also accepting. □

Lemma 8.2 (existence of accepting (reduced) runs) *For any AA and any input word, there exists an accepting run if and only if there exists a reduced accepting run.*

Proof Every reduced accepting run $\varsigma = \langle C, E\rangle$ can be trivially converted to a full accepting run: $\sigma = \langle C, E, \lambda\rangle$ where λ is the identity function.

Every accepting run $\sigma = \langle V, E, \lambda\rangle$ can be converted to a reduced accepting run by the following procedure. Apply Lemma 8.1 to remove a double occurrence of a state at a given level; remove vertices with no incoming edges; repeat until no suitable vertex pairs exist. The result is $\varsigma = \langle C, E'\rangle$ where $C = \lambda(V_0)\lambda(V_1)\ldots$ and $E'_i = \{\langle\lambda(v), q'\rangle \mid v \in V_i \wedge q \in \lambda(E_i(v))\}$. □

For the BMC encoding, we reduce the definition of runs further and consider simply sequences of configurations without recording the underlying DAG structure. Of course, a given configuration sequence may represent many different runs, which may not all be accepting. We therefore relate configuration sequences back to reduced runs to define the acceptance criterion.

Definition 8.3.4 (configuration sequence)
A configuration sequence C of an alternating co-Büchi automaton on a word $u_0u_1\ldots \in \Sigma^\omega$ is an infinite sequence $C_0C_1\ldots$ of configurations, where $C_i \subseteq Q$. As before, we have

$$C_0 \subseteq I$$

and successive configurations are related without reference to edges:

$$\forall q \in C_i, \exists\, \langle\alpha, s'\rangle \in \delta(q)\ .\ u_i \in \alpha \wedge s' \subseteq C_{i+1}$$

A configuration sequence is accepting if there exists a reduced run with the same configurations which is accepting.

For example, consider the run in Figure 8.1. The corresponding sequence of configurations is

$$\begin{aligned}
&\{\mathbf{G}(p \to \mathbf{F}\, q)\}\\
&\{\mathbf{F}\, q, \mathbf{G}(p \to \mathbf{F}\, q)\}\\
&\{\mathbf{F}\, q, \mathbf{G}(p \to \mathbf{F}\, q)\}\\
&\{\mathbf{F}\, q, \mathbf{G}(p \to \mathbf{F}\, q)\}\\
&\{\mathbf{G}(p \to \mathbf{F}\, q)\}
\end{aligned}$$

8.3.1.2 Superset Property of Runs

Both formulations of runs describe the minimal elements (or multiset of elements) of states at each point in time, but neither requires that the set consists solely of these elements. We may, without changing the language accepted, replace C_i with a superset of C_i (similarly V_i) provided that successive configurations (levels of the tree) can be modified to accommodate the evolution of the extra states while remaining consistent with the definitions of the runs. This is crucial to the encoding described below: we need only constrain the current configuration to be any superset of that described by the transitions.

8.3.2 BMC with Alternating Automata

The encoding of alternating automata is very similar to Büchi automata. Since a run is a sequence of configurations rather than states we use one atomic proposition to represent each state; configurations are then represented by conjunctions of states.

Given a VWAA representing LTL formula f, $\mathcal{A}_\phi = \langle Q, \Sigma, \delta, I, F\rangle$, we encode the presence of a a state q in the ith configuration by the proposition q^i. A configuration is encoded as a conjunction of its members: we write $[\![C]\!]^i = \bigwedge_{q\in C} q^i$, with $[\![\emptyset]\!]^i = \bot$. Note that this constrains the necessary, but not sufficient, members of the configuration, and so describes the smallest configuration that describes the run as discussed in Section 8.3.1.2. The targets of transitions can be seen as subsets of configurations and are hence encoded in the same way.

For VWAAs derived from LTL formulae as above, the transitions are labelled with a set of sets of atomic propositions: the set of permitted assignments to propositions. These can be denoted[3] by a conjunction of literals where $p \wedge q$ denotes $\{\alpha \mid \alpha \in \Sigma \wedge p \in \alpha\} \cap \{\alpha \mid \alpha \in \Sigma \wedge q \in \alpha\}$. We write $[\![\hat{\alpha}]\!]^i$ for the conjunction of literals representing $\hat{\alpha} \in 2^\Sigma$ in the ith state—this is particularly convenient as the implementation of the LTL to VWAA conversion [53] produces these conjunctions directly.

As before, the transition relation is given as a series of constraints

$$T_{\mathcal{A}_\phi}(i) = \bigwedge_{q\in Q}\left(q^i \rightarrow \bigvee_{\langle\hat{\alpha},s\rangle\in\delta(q)} \left([\![\hat{\alpha}]\!]^i \wedge [\![s]\!]^{i+1}\right)\right)$$

[3] See Remark 2 in Gastin and Oddoux [53].

and the initial set of configurations is encoded

$$I_{\mathcal{A}_\phi} = \bigvee_{C_0 \in I} [\![C_0]\!]^0$$

A VWAA run is accepting if no branch contains an infinite number of states from F. This can be assured on a k-prefix path if the empty configuration is reached at any point: the superset property means that there is a run in which successive configurations are also empty and hence no state is visited infinitely often. This also means that we can reduce the check to an empty kth configuration: this will hold even if the first empty configuration is before k.

$$P_{\mathcal{A}_\phi}(k) = \bigwedge_{q \in Q} \neg q(k)$$

For the loop case, we cannot simply check for an infinite number of occurrences of the members of F as the co-Büchi condition is on paths through the configuration space. That is, an accepting run could consist of an infinite number of paths each with a finite number of occurrences of an acceptance state. In this case the acceptance state would appear in a configuration within the loop suggesting that the state was visited infinitely often. In fact, we must make use of the *very weak* condition again: the only loops in VWAAs are self-loops, and hence the only paths that visit a state infinitely often must do so by always taking the self-loop transition. By the left-append and prefix closed property of accepting paths, we can deduce that if it is possible to take a non-self-loop transition from an accepting state then that state may be part of an accepting path.

$$F_{\mathcal{A}_\phi}(k, l) = \bigwedge_{q \in F} \bigvee_{i=l}^{k} \left(q^i \rightarrow \bigvee_{\substack{\langle \hat{\alpha}, s' \rangle \in \delta(q) \\ q \notin s'}} \left([\![\hat{\alpha}]\!]^i \wedge [\![s']\!]^{\rho_0(i+1)} \right) \right)$$

The correctness of this encoding follows from the following lemma.

Lemma 8.3 (acceptance of configuration sequence with self-loop exit) *A configuration sequence C for an AA on a word $u_0 u_1 \ldots \in \Sigma^\omega$ is accepting if, for every state $q \in F$ which occurs infinitely often, a non-self loop transition*

$$\exists i \,.\, \langle \hat{\alpha}, s' \rangle \in \delta(q) \wedge q \notin s' \wedge u_i \in \hat{\alpha} \wedge q \in C_i \wedge s' \subseteq C_{i+1}$$

may be taken infinitely often.

Proof By Definition 8.3.4, a configuration sequence is accepting if a reduced run exists over the same configurations. We demonstrate that such a reduced run can be constructed. For each state $q \in F$ which occurs infinitely often in the configuration sequence, choose infinitely many configurations C_i such that $q \in C_i$ (this need not be every configuration in which q occurs). The edge sets E_i are constructed such that a non-self-loops are described for q. The remaining states in C_i are connected by edges according to δ as described in Definition 8.3.3.

The remaining edges can be filled in in such a way that the resulting reduced run is accepting: successive configurations are constrained to be connected by transitions according to Definition 8.3.4 so at least the constructing sequence of edges exists; every state in F which occurs infinitely often in the sequence of configurations occurs only finitely often on each branch of the resulting reduced run because of the restriction on edges given above. □

As before, we give the encoding in terms of the general BMC formulation of Section 5.3.2:

$$\begin{aligned}
\mathrm{enc}_c^{\mathcal{A}}(\phi, k) &= I_{\mathcal{A}_\phi} \wedge \bigwedge_{i=0}^{k-1} T_{\mathcal{A}_\phi}(i) \\
\mathrm{enc}_n^{\mathcal{A}}(\phi, k) &= P_{\mathcal{A}_\phi}(k) \\
\mathrm{enc}_l^{\mathcal{A}}(\phi, k, l) &= F_{\mathcal{A}_\phi}(k, l) \wedge \bigwedge_{q \in Q} q^l \leftrightarrow q^k
\end{aligned}$$

Lemma 8.4 (correctness of encoding) *The encoding described by the expressions* $\mathrm{enc}_c^{\mathcal{A}}(\phi, k)$, $\mathrm{enc}_n^{\mathcal{A}}(\phi, k)$ *and* $\mathrm{enc}_l^{\mathcal{A}}(\phi, k, l)$ *is satisfied only when an accepting run exists for the AA* $\mathcal{A}_\phi$.

Proof By Lemmas 8.3 and 8.1, every sequence of configurations satisfying this encoding in the loop case can be expanded to an accepting run of the alternating automaton. In the prefix case, every satisfying sequence of configurations ends with the empty configuration so, by Lemma 8.1, can be expanded to an accepting run of the alternating automaton. □

This encoding produces a linear number of atomic propositions in the size of the LTL formula. The resulting propositional formula is linear in the product of the number of transitions and k, again except for $F_{\mathcal{A}_\phi}$ which is quadratic in k.

8.4 Alternating Automata and SNF

We have examined two established methods of encoding LTL for bounded model checking and introduced a third: the encoding via alternating automata. We now clarify the relationships and relative advantages of the encodings.

The configuration view of alternating automata makes it apparent that Fixpoint and AA are nearly equivalent. Step rules in SNF relate states and their successors to the evolved state of the model, while AA transitions relate states and their successors to the present state of the model. We can project each variable v created during LTL conversion to SNF to a VWAA state $\mathbf{X}\,v$: the set of SNF variables is directly related to the members of the configurations of the VWAA. Furthermore, we can show that SNF step rules created from LTL always have atomic antecedents (see 5.1): a necessary condition to relate step rules to transitions.

The condition used in the linear SNF encoding to represent eventualities corresponds to an assertion that x occurs finitely, not infinitely, often. It is introduced for the same states that, in the alternating automaton conversion, would be in the co-Büchi acceptance set. The difficulty of checking the co-Büchi acceptance condition is sidestepped by the start-of-loop projection introduced in Section 5.3.3. Effectively, all branches of the run are collapsed into one.

In fact, this is the main advantage of SNF over VWAAs: the encoding of the acceptance set is complex and comparatively large for the alternating automaton encoding. There are other advantages: not being a transition system, the variables introduced by SNF are not included in the loop-back condition L_k, eliminating the need for the empty-configuration assertion in the finite case. This can even reduce slightly the bound at which counterexamples are found. Alternating automata do benefit from the simplification [53] and simulation [49] reductions, some of which do not project directly to SNF; the advantages of these have the potential to outweigh the drawbacks of the encoding.

To demonstrate some of the differences between the approaches we give a selection of experimental results comparing a variety of BMC encodings. The existing encodings, the original BMC encoding [12] (marked dfl in the results), the SNF encoding and its refinement (snf and lin) are compared against Büchi

automata, in this case the Etessami and Holzmann [45] procedure (TMP), and the VWAA produced by the tool from Gastin and Oddoux [53] with and without its simplifications (aa and aa-).

To provide a comparison over a range of LTL specifications we fix the model for the experiments, using the distributed mutual exclusion example (see Section 7.4), at several bounds to illustrate scalability. The number and nesting depths of temporal operators appearing in the specifications are reported as pairs of numbers alongside their names in the tables. We used a modified version of NuSMV [22] with the compact CNF conversion (see Chapter 6; timings were made in the SAT solver zChaff [79].

The Table 8.1 shows three correct specifications, verified at bound 50. Rather than report the number of states that each automata conversion produces, we report the size of the CNF result. This means that the automaton methods can be directly compared to the SNF and direct encodings.

We observe that as the specifications become more complex, the simplicity of the SNF encoding has an increasing advantage. The alternating automata approach lags close behind the Büchi automata produced by TMP: a particularly interesting result, as the latter includes advanced simulation-based simplification techniques, while the former uses simple transition and state simplifications.

We illustrate the effect of the different encodings on counterexample size by comparing two incorrect specifications with different minimal counterexamples (Table 8.2). Here we see that the Büchi automaton procedure is slower due to the longer counterexample produced. The other procedures are all comparable although the VWAA method is slightly faster on the larger example.

8.5 Related Work

We have discussed the relationship between SNF and automata and observed that SNF appears to be most closely related to alternating automata. Two papers are available which take this idea further and give general translations between Büchi automata and SNF, and between alternating automata and SNF.

Bolotov, Fisher, and Dixon [16] describe a conversion from ω-automata (a class of which Büchi automata are a member) to SNF (the reverse conversion follows easily from the standard LTL to Büchi automaton conversions). The

Table 8.1: Comparison of automata and SNF techniques for BMC

Enc.	Bound	Clauses	Vars	Time	Clauses	Vars	Time	Clauses	Vars	Time
		Accessibility (4,2)			Overtaking 1 (5,5)			Overtaking 2 (8,8)		
aa	30	14596	2480	0.35	15737	2511	0.17	16339	2573	0.40
	40	19436	3280	1.47	20957	3321	2.02	21759	3403	1.05
	50	24276	4080	4.67	26177	4131	8.45	27179	4233	10.11
aa-	30	15325	2759	0.39	17011	2945	0.35	18065	3069	0.35
	40	20405	3649	1.39	22651	3895	1.40	24055	4059	1.35
	50	25485	4539	5.28	28291	4845	11.87	30045	5049	7.23
snf	30	14298	2418	0.97	14481	2480	0.23	14814	2573	0.25
	40	19038	3198	0.90	19281	3280	0.75	19724	3403	1.08
	50	23778	3978	4.04	24081	4080	2.76	24634	4233	2.22
lin	30	14299	2449	0.72	14483	2511	0.21	14816	2604	0.27
	40	19039	3239	0.89	19283	3321	0.96	19726	3444	0.88
	50	23779	4029	4.43	24083	4131	4.17	24636	4284	2.59
TMP	30	14599	2418	0.75	16559	2449	0.42	17898	2480	0.46
	40	19439	3198	4.54	22049	3239	1.43	23828	3280	1.24
	50	24279	3978	4.90	27539	4029	3.54	29758	4080	7.07
dfl	30	15848	2356	0.26	41874	2356	0.37	Encoding > 1800 secs		
	40	21908	3116	1.47	81539	3116	1.92			
	50	28368	3876	9.83	142404	3876	17.69			

Table 8.2: Further results

Enc.	k	Time	k	Time
	Priority 1 (4,2)		Priority 2 (4,2)	
aa	14	0.03	53	0.30
aa-	14	0.03	53	0.89
snf	13	0.02	52	0.49
lin	13	0.02	52	0.83
TMP	53	3.26	> 200	
dfl	13	0.02	52	1.15

conversion is based on an existing conversion from Büchi automata to QPLTL (this may be considered an extension of QLTL to include past-time operators). A variable is introduced for each state of the automaton. An accepting run is then encoded in the expected way, by forming the conjunction of the initial states, the transition relation (in the form $\mathbf{G} \bigvee_{q,q' \in S} q \wedge \mathbf{X} q' \wedge \langle q, q' \rangle \in \delta$), and the acceptance condition (in the form $\mathbf{G}\,\mathbf{F} \bigvee_{q \in F} q$). This is converted to a set of SNF rules consisting of a transition rule for each state ($q_i \Rightarrow \mathbf{X}\,\delta(q)$), a global assertion that the system is in exactly one state at a time, and a set of rules following from the acceptance condition. This conversion results in $O(n^2)$ rules being defined for an automaton of n states.

Dixon, Bolotov, and Fisher [39] describe a conversion from alternating automata to SNF and vice versa, although they use a Büchi, rather than co-Büchi, acceptance condition. The same observation is made in the paper as we make in this chapter: "Sets of $\mathrm{SNF}_{\mathrm{PLTL}}$ clauses are intuitively similar to the transition functions of alternating automata". Rather than conflate all states at a given point in the run to form configurations as described above, Dixon et al. distinguish between occurrences of a state before and after an accepting state. Two mutually exclusive variables in the SNF are used to represent each state in the AA, indicating whether that state is current and follows an accepting state, or that it is current but does not follow an accepting state. The acceptance condition is simply $\mathbf{G}\,\mathbf{F} \neg e$ where e is the disjunction of all non-following-acceptance state variables. The encoding of the transition relation is concerned with switching between the two sets of variables according

to whether accepting states have been seen. This conversion results in a polynomial number of rules in the size of the AA. In contrast with the BMC encoding given above, much of the complexity of the acceptance condition has been pushed into the transition relation.

In the reverse direction, Dixon et al. give a refinement of the LTL to AA conversion defined on SNF directly. Fewer states are produced by this conversion: rather than using a state for every temporal subformula, the following states are defined: one for the whole formula; one for the whole formula excluding initial rules (this is also the only accepting state); one for each step rule; one for each eventuality rule. The state representing the whole formula excluding step rules has a self loop transition which also leads to the conjunction of the individual rules: this models the outermost **G** of SNF.

The main difference between these papers and the problems considered in this chapter are that we restrict ourselves here to automata and SNF which is derived from LTL. This means that we make use of the 'very weak' property, and we see a much closer correspondence between SNF and alternating automata due to the restricted form of SNF described in Lemma 5.1. These papers, however, treat general SNF (with disjunctions in the antecedents, hence preventing SNF rules from being compared directly to AA transitions) and general alternating automata (with arbitrary loops). This is the root of the polynomial complexity in the conversions given, where we are able to describe above a linear size correspondence.

8.6 Summary and Conclusions

Int his chapter we have given an overview of techniques for converting LTL to automata. The LTL to Büchi automaton conversion has a long history of applications in model checking, and required for example to extend symbolic model checking to LTL properties. The LTL to alternating automaton approach is more recent, and has emerged as a preprocessing step before conversion to a Büchi automaton.

We have described BMC encodings based directly on both Büchi automata and alternating automata. The former is a straightforward extension of the conversion of Kripke structures to propositional logic described in Chapter 3. The latter is made more complex by the acceptance condition which requires

that no accepting states in and accepting run of the alternating automaton may appear infinitely often in any path. Since this does not preclude the state appearing infinitely often overall (since infinitely many paths may be required to describe a run) the encoding of the acceptance condition is based on the 'very weak' property which restricts loops in the AA to self-loops. This encoding of the acceptance condition is the significant contribution of this chapter.

As a result of exploring these encodings, we find that the SNF expressions which result from the transformation of LTL are very closely related to the AA derived from the same LTL. The main difference stems from the partial unrolling of the fixpoint described in Section 4.4.3.

The main advantage of automata-based bounded model checking, the high state of development of the conversion procedures, is balanced by the numerous drawbacks of conversion. We have described how the use of alternating automata overcomes many of these problems and demonstrated their use for BMC. A simple alternating automata encoding has been shown to be almost as effective as a highly developed Büchi automata approach, although both lag behind the SNF encoding (without any simplification) on many of the examples given.

Chapter 9

Conclusions

We review here the contributions made by this thesis to the field of bounded model checking and draw conclusions on the work.

9.1 Review of Thesis

This work has been concerned with changes to the BMC procedure given by Biere, Cimatti, Clarke, and Zhu [12] in an attempt to improve its performance. The most significant contribution has been the development of a series of SNF-based encodings for BMC (Chapter 3), one of which is linear size in the size of the input formula as well as being significantly faster in practice.

In the process we have made a series of other, smaller contributions. The BMC encoding defined by Biere et al. has been presented in a new form (Chapter 3) which makes the link between the infinite and bounded semantics clearer. In this presentation we proposed a simple hierarchy of semantics to capture the possible relationships between the infinite and bounded interpretations of LTL. The form of the SNF presentation (Chapter 4), and the unification of top-down and bottom-up conversion procedures (Section 4.5)) is a significant extension to the presentations of Bolotov [14], Dixon [40], Fisher [46] and others.

The new presentation of the SNF encoding in Chapter 5 is a significant improvement in rigour and clarity over the previously published presentations [24, 48]. The relationship between SNF and alternating automata (Chapter 8) helps place the SNF in context with regards to other LTL model checking approaches.

We have presented a new clause form conversion for RBCs (Chapter 6)

which takes linear time to execute, while producing more compact results than other linear procedures, and optimally small results when the input is a linear tree. In addition, we have shown how other conversion procedures such as those of Boy de la Tour [17, 18] and Plaisted and Greenbaum [84] can also be projected to RBCs.

9.2 Conclusions

This thesis has focused on two particular changes to the encoding used in BMC.

Firstly, the use of SNF as an intermediate representation has been demonstrated to significantly reduce the time spent in the SAT solver for all BMC examples, as well as producing the expected reductions in the numbers of clauses and propositions in the problems. Several encodings were proposed, and two in particular were benchmarked: a quadratic encoding and a linear encoding.

The most interesting result has been that on most examples, the quadratic encoding performed better. We conjecture that the additional complexity of the linear encoding, which includes additional renamings of some subformulae, increases the solving time more than the additional propositions and copies of like subformulae introduced by the quadratic encoding.

Although the transformation to SNF is itself intuitive and straightforward, as its encoding, the clear and formal presentation of these steps has turned out to be rather more complex. By turning to the denotational semantics of QLTL, the presentation has been made more rigorous.

For an alternative perspective to the experimental results, and to further understand the normal-form-based encodings of the specification, we have examined the standard automata-based techniques for model checking linear-time logics. Automata turn out to be amenable to use as BMC encodings, and a brief experimental comparison suggests that there is significant scope in exploring this avenue of research. SNF and alternating automata in particular are more closely related than the literature would suggest, in the case that they are both used to represent LTL formulae.

The new *compact* clause form conversion for RBCs also has a significant part to play in the results: indeed, much of the reduction in time spent in the

SAT solver is attributable to the clause form conversion. This conversion is also significant as the same restricted optimality applies to it as to the Boy de la Tour conversion, but it is linear-time in the size of the input formula rather than quadratic-time. Its efficacy with respect to other available clause form conversions is clearly borne out by the results.

Chapter 10

Future Work

In this thesis we have outlined just one possible approach to one of the encoding problems in BMC. In many ways, we have only scratched the surface with regards to possible improvements to the BMC procedure, and we describe here a number of promising directions in which we feel the work could be taken next.

10.1 Encoding the model

This thesis has addressed the problem of encoding the temporal logic specification the problem of obtaining an efficient model encoding has received significantly less attention. In NuSMV [22] a common front end is used for both symbolic and bounded model checking approaches, working in terms of BDDs [20] which are then transformed to propositional logic.

We consider three particularly interesting avenues in which this aspect of the encoding could be developed.

10.1.1 BDD to RBC conversion

Binary decision diagrams (BDDs) are a type of DAG representation of propositional formulae, orthogonal to RBCs (see Section 2.3.1) in the sense that internal nodes represent variables and edges their possible assignments. The leaves of BDDs give the outcome of the evaluation of the formula.

If a BDD representation of the model is to be used in the early stages of the model checker, the representation will eventually have to be converted to

an RBC in preparation for CNF conversion (see Chapter 6). This conversion is straightforward: for a BDD X of the form shown in Figure 10.1(a), the propositional representation $t(X)$ is $(a \rightarrow t(X)) \wedge (\neg a \rightarrow t(Y))$—as an RBC this would be three vertices.

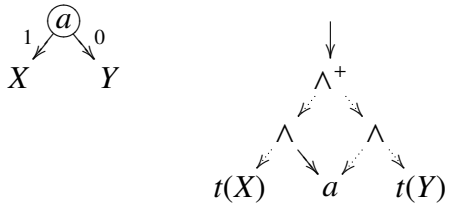
However, there are important cases in which this transformation is inefficient. In particular, BDD representations of arithmetic tend to include exclusive-or expressions, as shown in Figure 10.1(b), and more generally in Figure 10.1(c). In each case we have given the expression produced by the transformation above, and an alternative representation which may be more compact. This technique of matching larger patterns in the BDD to generate a better propositional formula could be generalised to other characteristic patterns. This approach also needs to be verified experimentally—as the patterns become larger, it becomes harder to be sure that the RBC is indeed preferable.

10.1.2 Specialised Propositional Encodings

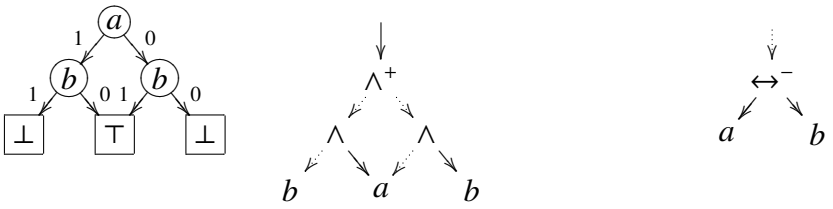
Specialised SAT encodings continue to be published in diverse fields, and sometimes yield novel and interesting encodings that could be put to use in BMC. One such encoding, proposed by Bailleux and Boufkhad [5] for cardinality constraints (constraints on the number of atomic propositions in a given set which are assigned true) actually defines a general encoding for base-1 (unary) addition which has the potential for being extended to other operators.

Cumulative unary is a variant of the one bit per value encoding of integers. Consider a set of integer variables $x_1, x_2, \ldots$ representing integer n. The cumulative integer interpretation of a variable x_i is $n \geq i$. That is, to represent an integer n for some maximum m, $0 \leq n < m$, we require $x_1 \wedge x_2 \wedge \cdots \wedge x_n \wedge \neg x_{n+1} \wedge \neg x_{n+2} \wedge \cdots \wedge \neg x_m$. The main advantage of this encoding over standard unary is that inequality can be established with a linear, rather than quadratic, number of clauses. Also, like with the binary encoding, all-different and at-least-one clauses (quadratic size in the number of bits used) are not required to preserve correctness.

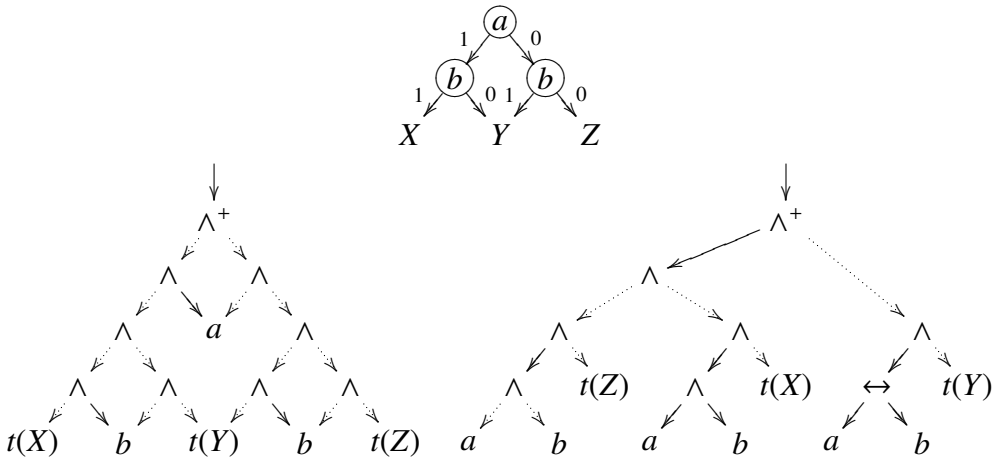
Cumulative unary is used by Bailleux and Boufkhad [5] to encode cardinality constraints using a tree of summations of increasing length. They are able to show that their encoding of summation can be solved using unit



(a) General BDD to RBC translation: $(a \rightarrow t(X)) \wedge (\neg a \rightarrow t(Y))$



(b) Simple XOR: $(a \rightarrow \neg b) \wedge (\neg a \rightarrow b)$ or $\neg(a \leftrightarrow b)$



(c) General XOR:

$(a \rightarrow (b \rightarrow t(X) \wedge \neg b \rightarrow t(Y))) \wedge (\neg a \rightarrow (b \rightarrow t(Y) \wedge \neg b \rightarrow t(Z)))$

or $((\neg a \wedge \neg b) \rightarrow t(Z)) \wedge ((a \wedge b) \rightarrow t(X)) \wedge (\neg(a \leftrightarrow b) \rightarrow t(Y))$

Figure 10.1: Examples of BDD transformations for XOR. From left to right: BDD, unoptimised RBC, optimised RBC

propagation under certain circumstances (sufficient constraints on the input and output propositions to establish a unique result). The encoding for the addition $r = a + b$ where $0 \leq a \leq m_a$, $0 \leq b \leq m_b$, $0 \leq r \leq m$, is given as

$$\bigwedge_{\substack{0\leq\alpha\leq m_a\\0\leq\beta\leq m_b\\0\leq\sigma\leq m\\\alpha+\beta=\sigma}} (a_\alpha \wedge b_\beta \rightarrow r_\sigma) \wedge (\neg a_{\alpha+1} \wedge \neg b_{\beta+1} \rightarrow \neg r_{\sigma+1})$$

We can view the implications as $a \geq \alpha \wedge b \geq \beta \rightarrow r \geq \sigma$ and $a \leq \alpha \wedge b \leq \beta \rightarrow r \leq \sigma$ for all α, β such that $\alpha + \beta = \sigma$. A pair of implications fixing the maximum and minimum conditions for the result is significantly more efficient that the equivalent iff formulation

$$\bigwedge_{0\leq\sigma\leq m} \Big(\bigvee_{\substack{0\leq\alpha\leq m_1\\0\leq\beta\leq m_2\\\alpha+\beta=\sigma}} a_\alpha \wedge b_\beta \Big) \leftrightarrow r_\sigma$$

where the 'if' direction generates a huge number of clauses with terms made redundant by the cumulative nature of the integer representation. In order to generalise the use of cumulative unary we must preserve the ability to generate efficient CNF in this way.

The addition operation defined above must introduce a new set of propositions each time it is used. This seems like a good trade-off: the construction iterates over each combination of the input propositions leading to $O(n^2)$ clauses; if addition were performed over more variables, the number of clauses would similarly rise—$O(n^k)$ for k variables—as would the length of the clauses.

We therefore suggest adding more functionality to the single operation. That is, any operations that can be performed without a significant cost in terms of the number of clauses can be performed as part of the addition operation. This obviously requires a bit of extra work to rearrange the arithmetic expression to the preferred form for the operations available, but there are standard compilation techniques for this.

We identify the following cheap operations:

Constant addition is available by inserting the requisite number of $\top$s into the result.

Constant multiplication of the addends is performed by repeating each bit the appropriate number of times.

Constant integer division of the addends is performed by selecting only regularly spaced bits.

Constant modulo that is, finding the remainder after integer division, is performed by a series of tests and disjunctions which can be incorporated into the clause set.

To maximise generality, we suggest the following form for generalised addition:

$$\left(\frac{m_\alpha}{d_\alpha}(\alpha + i_\alpha) + \frac{m_\beta}{d_\beta}(\beta + i_\beta) + i_\sigma\right) \mod d_\sigma + j_\sigma = \sigma$$

The addends i_α and i_β are required separately as constant addition may change the result of the integer division. For consistency, we assume that multiplication by m is always performed *before* division by d, allowing for the greatest numerical stability. The two result addends i_σ and j_σ are both required as addition does not commute with modulo.

Apart from the modulo operation, implementation is as simple as changing the final condition on the conjunction:

$$\bigwedge_{\substack{0\le\alpha\le m_1\\ 0\le\beta\le m_2\\ 0\le\sigma\le m\\ \left(\left\lfloor\frac{m_\alpha}{d_\alpha}(\alpha+i_\alpha)\right\rfloor+\left\lfloor\frac{m_\beta}{d_\beta}(\beta+i_\beta)\right\rfloor=i_\sigma\right)}} (a_\alpha \wedge b_\beta \to r_\sigma) \wedge (\neg a_{\alpha+1} \wedge \neg b_{\beta+1} \to \neg r_{\sigma+1})$$

To perform modulo k, a variable γ of n bits is divided into blocks, each k bits long. The first bit of each block corresponds to 0, and is discarded. The result is the bit-wise disjunction of all of the blocks. The condition for each block to be used is $(i-1)k \le \gamma < ik$ for the ith block. The comparison $\gamma < ik$ turns out to be the zeroth bit of each block. For example, consider computing $g \mod 3$. The result is two bits long:

$$\begin{aligned}
&\gamma_1 \wedge (\gamma_3) \quad \vee \quad \gamma_4 \wedge (\neg\gamma_3 \wedge \gamma_6) \quad \vee \quad \gamma_7 \wedge (\neg\gamma_6 \wedge \gamma_9) \quad \ldots \\
&\gamma_2 \wedge (\gamma_3) \quad \vee \quad \gamma_5 \wedge (\neg\gamma_3 \wedge \gamma_6) \quad \vee \quad \gamma_8 \wedge (\neg\gamma_6 \wedge \gamma_9) \quad \ldots
\end{aligned}$$

We require one bit to store the each of the intermediate flag values for the $\gamma < ik$ comparisons, and one bit for each bit of the result. The total number of bits required is therefore $k - 1 + \left\lfloor\frac{g}{k}\right\rfloor$. We represent this in the encoding by choosing the result bit with the function $m(i)$ defined as

$$m(i,k) = \begin{cases} i & \text{if } i \mod k = 0 \\ i \mod k & \text{otherwise} \end{cases}$$

with the functions $ub(i)$ and $lb(i)$ giving the upper and lower bound bits for the block containing a given bit:

$$ub(i) = k\left\lfloor \frac{i}{k} \right\rfloor \qquad lb(i) = k\left\lceil \frac{i}{k} \right\rceil$$

The encoding becomes

$$\bigwedge_{\substack{0\le\alpha\le m_1\\ 0\le\beta\le m_2\\ 0\le\sigma\le m\\ \left(\left\lfloor \frac{m_\alpha}{d_\alpha}(\alpha+i_\alpha)\right\rfloor+\left\lfloor \frac{m_\beta}{d_\beta}(\beta+i_\beta)\right\rfloor=i_\sigma\right)}} \begin{array}{c}(a_\alpha \wedge b_\beta \wedge \neg r_{lb(\sigma)} \wedge r_{ub(\sigma)} \rightarrow r_{m(\sigma,d_\sigma)})\wedge\\ (\neg a_{\alpha+1} \wedge \neg b_{\beta+1} \wedge \neg r_{lb(\sigma)} \wedge r_{ub(\sigma)} \rightarrow \neg r_{m(\sigma+1,d_\sigma)})\end{array}$$

Despite of its apparent complexity, this representation retains the properties that led us to choose cumulative unary. In particular, the whole multiply/divide/add/modulo block is still computed by a number of unit propagations proportional to the size of the input words.

10.1.3 Encodings Inspired by Digital Electronics

A final approach to improved model encodings is to draw on the established field of digital electronics. A variety of techniques and designs have emerged for arithmetic and other circuits, optimised variously for speed (fewest gates on the critical path between input and output), size (fewest gates overall), and other criteria. One interesting approach would be to compare the performance of a several different adder circuits, for example, to determine how the criteria used in the digital electronics domain correspond to those for the high performance of a SAT solver.

The BDD approach of NuSMV actually produces a circuit very similar to a standard ripple-carry adder. This has the disadvantage that high-order bits cannot be computed until the carry has propagated from the lower-order bits. In a SAT domain, this corresponds to a large number of assignments being required before the value of the final bit is fully constrained. Conversely, a carry-propagate adder, while more complex, uses a separate sub-circuit to pre-compute all of the carry values. In SAT this could result in a shorter chain of assignments before a given bit is constrained.

10.2 Development of the Semantics

The derivation of explicit semantics in Chapter 3 raises questions about their origin, and whether they are the best performing semantics possible. We introduced a hierarchy of semantics, but have not developed the concept further. Some possible directions for research in this field are described below.

10.2.1 Maximality of Semantics

In Chapter 3 we introduce a partial order for the completeness and soundness of the bounded (finite prefix and k-loop) semantics. By restricting the domain of the semantics to *context-free* semantics we are able to find a straightforward semantics that is maximal in the sense defined. However, there are clear cases where considering a non-context sensitive semantics can bring about an improved encoding. For example, consider the equality

$$\mathbf{F}\,\mathbf{F}\,f = \mathbf{F}\,f$$

A semantics which equates these two expressions would be expected to have a simplifying effect on the end encoding.

10.2.2 Exploiting the Complete Semantics

In Chapter 3 we focused on the sound semantics as the most appropriate for bug hunting: a witness path found in the bounded semantics indicates the existence of a witness path in the infinite semantics. There is a case for using the alternative, complete, semantics: the absence of a witness in the bounded case indicates the absence of one in the infinite case, and so the process can be used to demonstrate that a model is bug-free.

Sheeran, Singh, and Stålmarck [88] give an explicit encoding for the case of $\mathbf{G}\,f$ which is in fact a combination of the complete and sound semantics. The bound is increased iteratively, and the presence of bugs is checked using the sound semantics; the complete semantics is used to determine when to stop searching for bugs.

10.3 Automata-Inspired Approached

There is a considerable body of work concerning the conversion from LTL to automata, which we began to discuss in Chapter 8. Many lessons can be learnt from these approaches.

10.3.1 LTL Simplifications

The LTL simplification techniques are crucial to the good performance of LTL to Büchi automata conversions. However, no attempt has been to incorporate these techniques in BMC. Furthermore, the changed semantics of LTL for the paths types in BMC may suggest additional simplifications that would not be available in the infinite semantics.

10.3.2 Alternating Automata

In Chapter 8 we described an encoding for alternating automata to propositional logic for use in BMC. For much of the encoding, the results obtained are very similar to those obtained from the SNF encoding. The main difference comes from the encoding of the acceptance condition: the corresponding problem for SNF, encoding eventualities, forms the bulk of Chapter 5.

It may prove possible to avoid the complexity of the acceptance condition by first converting an alternating automaton to SNF, then using the SNF encoding given in Chapter 5. Similarly, an SNF-inspired encoding for the acceptance condition may be simpler; we conjecture that a linear space encoding is possible, just as for SNF.

Improving the AA approach to the point where it is directly competitive with SNF opens up the possibility of exploiting the advanced simplification techniques available for alternating automata. such as those from Fritz [49].

10.3.3 Direct Application of the Fixpoint Transformations

In the derivation of SNF transformations given in Chapter 4 we focus on obtaining the form of SNF as given by Fisher [46] and Bolotov [14]. However, in order to obtain the specific form, we are forced to derive suitably adapted fixpoint expressions (Figure 4.2). An alternative approach, as proposed by Jackson [64] is to abandon the form of SNF but retain the structure of the

transformations. The following identities make use of context functions (see Section 2.5.3) and together define a transformation which results in a formula of the form $\mathbf{G}(\phi_0) \wedge \mathbf{G}(\phi_1) \wedge \cdots$ where ϕ_i are formulae containing only propositional operators and $\mathbf{X}$ or $\mathbf{F}$; this is a generalisation of the form of SNF.

$$\begin{aligned}
\Psi[\mathbf{G}\,\phi] &= \Psi[x] \wedge \mathbf{G}(a \rightarrow \phi \wedge \mathbf{X}\,a) \\
\Psi[\phi\,\mathbf{U}\,\psi] &= \Psi[x \wedge \mathbf{F}\,\psi] \wedge \mathbf{G}(a \rightarrow \psi \vee (\phi \wedge \mathbf{X}\,a)) \\
\Psi[\phi\,\mathbf{R}\,\psi] &= \Psi[x] \wedge \mathbf{G}(a \rightarrow \psi \wedge (\phi \vee \mathbf{X}\,a))
\end{aligned}$$

These transformations need to be explored experimentally since the resulting formulae are structurally quite different from SNF. However, the advantage is that the form of the transformations is simpler both to prove and to implement. In addition, the procedure becomes even closer to the alternating automata construction given in Chapter 8.

Appendix A

Publications

We include here the peer-reviewed papers relevant to this thesis which have been accepted for publication during the course of the PhD work, indicating where appropriate the contribution made by the PhD candidate.

A.1 A fixpoint based encoding for bounded model checking [48].

Candidate conceived and wrote the entire paper; the contribution of other authors was advisory and methodological.

A.2 Bounded verification of Past LTL [24].

Candidate devised and implemented the conversion approach; experiments were undertaken by the other authors; the paper consisted of equal contributions from all authors.

A.3 Clause form conversions for Boolean circuits [63].

Candidate devised and implemented the conversion and wrote the paper; the proofs which form the main body of the paper grew out of discussions between the authors.

A.4 Bounded model checking with SNF, alternating automata and Büchi automata [89].

Candidate worked alone.

A Fixpoint Based Encoding for Bounded Model Checking

Alan Frisch[1], Daniel Sheridan[1], and Toby Walsh[2]

[1] University of York, York, UK
{frisch,djs}@cs.york.ac.uk
[2] Cork Constraint Computation Centre, University College Cork, Cork, Ireland
tw@4c.ucc.ie

Abstract. The *Bounded Model Checking* approach to the LTL model checking problem, based on an encoding to Boolean satisfiability, has seen a growth in popularity due to recent improvements in SAT technology. The currently available encodings have certain shortcomings, particularly in the size of the clause forms that it generates. We address this by making use of the established correspondence between temporal logic expressions and the fixpoints of predicate transformers as used in symbolic model checking. We demonstrate how an encoding based on fixpoints can result in improved performance in the SAT checker.

1 Introduction

Bounded Model Checking (BMC) [2] is an encoding to Boolean Satisfiability (SAT) of the LTL model checking problem. The encoding is achieved by placing a bound on the number of time steps of the model that are to be checked against the specification. The resulting Boolean formula contains variables representing the state variables of the model at each step along a path, together with constraints requiring the path to be contained within the model and to violate the specification. The result of the SAT checker is thus a path in the model which is a counterexample to the specification, or failure, which means that no such path exists within the bound.

The encoding of the LTL specification in BMC is defined recursively on the structure of the formula. While for simple specifications this is sufficient, more complex specifications such as bounded existence and response patterns [7] lead to an exponential blowup in the size of the resulting Boolean formula. Recent improvements to the encoding in NuSMV [4] have not removed this restriction.

The fixpoint characterisations of temporal operators [8] have been exploited in other model checking systems such as SMV [14]; we discuss an approach to their use in an encoding of LTL for BMC which produces more compact encodings which can be solved more quickly in the SAT solver.

2 Bounded Model Checking

2.1 Background

A model checking problem is a pair $\langle M, f \rangle$ of a model and a temporal logic specification.

A model M is defined as a Kripke structure $\langle S, R, L, I \rangle$ where S is a set of states; $R : S \to S$ is the transition relation; $L : S \to \mathcal{P}(AP)$ is the labelling function, marking each state with the set of atomic propositions (AP) that hold in that state; and I is the set of initial states, which may be equal to S. A path $\pi \in M$ is a sequence of states $s_0, s_1, \ldots \in M$ such that $\forall i.(s_i, s_{i+1}) \in R$. We write $\pi(i)$ to refer to the ith state along the path.

The *model checking problem for LTL* is to verify that for an LTL formula f, for all paths $\pi_i \in M$ such that $\pi_i(0) \in I$, $(M, \pi_i) \models f$.

2.2 Path Loops

We say the a path π is a *k-loop* if for all $i \geq 0$, the $(k+i)$th state in π is identical to the $l+i$th state for some $l, 0 \leq l < k$. If a path is known to be a loop, it is possible to verify the correctness of infinite time specifications such as *always* ($\mathbf{G}$) by checking just the first k states in the path.

2.3 Boolean Satisfiability

Boolean satisfiability (SAT) is the problem of assigning Boolean values to variables in a propositional formula, in such a way as to make the formula evaluate to true (to *satisfy* the formula). For example, for the formula $(a \vee \neg b) \wedge (b \vee \neg c) \wedge (\neg c \vee \neg a)$ can be satisfied by *e.g.* the assignment $a = 1, b = 1, c = 0$.

SAT solvers derived from the Davis-Putnam algorithm [5] require input in clause form (CNF): a conjunction of *clauses*, each of which is a disjunction of literals.

A number of high performance SAT solvers are available, making SAT a convenient 'black box' back end for a number of different problems.

2.4 The Bounded Model Checking Encoding

The bounded model checking encoding represents k states along a bounded path π_{bmc} together with a conjunction of constraints requiring π_{bmc} to be a *valid path in M* and be a *counterexample of f*. The 'valid path' constraint is a propositional encoding of the transition relation. We can see from the bounded semantics of LTL (Figure 1) that there are two ways of violating each operator in the specification, depending on whether π_{bmc} is a k-loop; the 'counterexample' constraint is therefore a disjunction of the ways in which the specification may be violated.

We write the bounded model checking encoding of a problem with bound k, model M and specification f as

$$[\![M, \neg f]\!]_k$$

$$
\begin{array}{ll}
(M,\pi) \models_k^i a & \Leftrightarrow a \in L(\pi(i)) \quad \text{for atomic } a \\
(M,\pi) \models_k^i \neg f_1 & \Leftrightarrow (M,\pi) \not\models_k^i f_1 \\
(M,\pi) \models_k^i f_1 \wedge f_2 & \Leftrightarrow (M,\pi) \models_k^i f_1 \text{ and } (M,\pi) \models_k^i f_2 \\
(M,\pi) \models_k^i f_1 \vee f_2 & \Leftrightarrow (M,\pi) \models_k^i f_1 \text{ or } (M,\pi) \models_k^i f_2 \\
(M,\pi) \models_k^i \mathbf{X}\, f_1 & \Leftrightarrow \begin{cases} (M,\pi) \models_k^{i+1} f_1 & \text{if } \pi \text{ is a } k\text{-loop} \\ (M,\pi) \models_k^{i+1} f_1 \wedge i < k & \text{otherwise} \end{cases} \\
(M,\pi) \models_k^i \mathbf{F}\, f_1 & \Leftrightarrow \begin{cases} \exists j, i \le j.(M,\pi) \models_k^j f_1 & \text{if } \pi \text{ is a } k\text{-loop} \\ \exists j, i \le j \le k.(M,\pi) \models_k^j f_1 & \text{otherwise} \end{cases} \\
(M,\pi) \models_k^i \mathbf{G}\, f_1 & \Leftrightarrow \begin{cases} \forall j, i \le j.(M,\pi) \models_k^j f_1 & \text{if } \pi \text{ is a } k\text{-loop} \\ \bot & \text{otherwise} \end{cases} \\
(M,\pi) \models_k^i [f_1\, \mathbf{U}\, f_2] & \Leftrightarrow \begin{cases} \exists j, i \le j.(M,\pi) \models_k^j f_2 & \\ \quad \wedge \forall n, i \le n < j.(M,\pi) \models_k^n f_1 & \text{if } \pi \text{ is a } k\text{-loop} \\ \exists j, i \le j \le k.(M,\pi) \models_k^n f_2 & \\ \quad \wedge \forall n, i \le n < j.(M,\pi) \models_k^n f_1 & \text{otherwise} \end{cases} \\
(M,\pi) \models_k^i [f_1\, \mathbf{R}\, f_2] & \Leftrightarrow \begin{cases} \exists j, i \le j.(M,\pi) \models_k^j f_1 & \\ \quad \wedge \forall n, i \le n \le j.(M,\pi) \models_k^j f_2 & \text{if } \pi \text{ is a } k\text{-loop} \\ \exists j, i \le j \le k.(M,\pi) \models_k^j f_1 & \\ \quad \wedge \forall n, i \le n \le j.(M,\pi) \models_k^j f_2 & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 1. The Bounded Semantics of LTL

Table 1. The BMC Encoding for LTL

f	$[\![f]\!]_k^i$	${}_l[\![f]\!]_k^i$
$\mathbf{G}\, f_1$	$\bot$	$\bigwedge_{j=\min(i,l)}^{k} {}_l[\![f_1]\!]_k^j$
$\mathbf{F}\, f_1$	$\bigvee_{j=i}^{k} [\![f_1]\!]_k^j$	$\bigvee_{j=\min(i,l)}^{k} {}_l[\![f_1]\!]_k^j$
$\mathbf{X}\, f_1$	$i < k \wedge [\![f_1]\!]_k^{i+1}$	$\left(i < k \wedge {}_l[\![f_1]\!]_k^{i+1}\right) \vee \left(i = k \wedge {}_l[\![f_1]\!]_k^{l}\right)$
$f_1\, \mathbf{U}\, f_2$	$\bigvee_{j=i}^{k} ([\![f_2]\!]_k^j \wedge \bigwedge_{n=i}^{j-1} [\![f_1]\!]_k^n)$	$\bigvee_{j=i}^{k} ({}_l[\![f_2]\!]_k^j \wedge \bigwedge_{n=i}^{j-1} {}_l[\![f_1]\!]_k^n)$ $\vee \bigvee_{j=l}^{i-1} ({}_l[\![f_2]\!]_k^j \wedge \bigwedge_{n=i}^{k} {}_l[\![f_1]\!]_k^n \wedge \bigwedge_{n=l}^{j-1} {}_l[\![f_1]\!]_k^n)$
$f_1\, \mathbf{R}\, f_2$	$\bigvee_{j=i}^{k} ([\![f_1]\!]_k^j \wedge \bigwedge_{n=i}^{j} [\![f_2]\!]_k^n) \bigwedge_{j=\min(i,l)}^{k} {}_l[\![f_2]\!]_k^j \vee \bigvee_{j=i}^{k} ({}_l[\![f_1]\!]_k^j \wedge \bigwedge_{n=i}^{j} {}_l[\![f_2]\!]_k^n)$ $\vee \bigvee_{j=l}^{i-1} ({}_l[\![f_1]\!]_k^j \wedge \bigwedge_{n=i}^{k} {}_l[\![f_2]\!]_k^n \wedge \bigwedge_{n=l}^{j} {}_l[\![f_2]\!]_k^n)$	

Given the functions ${}_l\mathrm{L}_k(\pi)$ which holds when π is a k-loop with $\pi(k) = \pi(l)$ and $\mathrm{L}_l(\pi) = \bigvee_{l=0}^{k} {}_l\mathrm{L}_k$ which holds when π is any k-loop, the general translation is defined as[3]:

$$[\![M, f]\!]_k := [\![M]\!]_k \wedge \left(\left(\neg \mathrm{L}_k(\pi) \wedge [\![f]\!]_k^0\right) \vee \bigvee_{l=0}^{k} \left({}_l\mathrm{L}_k(\pi) \wedge {}_l[\![f]\!]_k^0\right) \right) \tag{1}$$

where $[\![M]\!]_k$ denotes the encoding of the transition relation of M as a constraint on π with bound k; $[\![f]\!]_i^k$ and ${}_l[\![f]\!]_i^k$ denote the encoding of the LTL formula f evaluated along path π at time i, where π is a non-looping path and a k-loop to l respectively. These encodings are given in Table 1. Biere et al. show the correctness of some of these encodings in [2]; we will not repeat their proofs here.

Theorem 1 in Biere et al. [2] states that bounded model checking of this form is complete provided that the bound k is sufficiently large.

3 Exploiting Fixpoints in BMC

The approach that we have taken to making a fixpoint-based encoding for BMC is based on a clause-style normal form for temporal logic. After converting the specification to this form, we can redefine the encoding to specifically take advantage of the properties of the normal form.

3.1 The Separated Normal Form

Gabbay's Separation Theorem [11] states that arbitrary temporal formulæ may be written in the form $\mathbf{G}\,(\bigwedge_i (P_i \Rightarrow F_i))$ where P_i are (strict) past time formulæ and F_i are (non-strict) future time formulæ.

Fisher [9] defines a normal form for temporal logic based on the Separation Theorem and gave a series of transformations for reaching it. The general form of SNF is the same as the separation theorem; the implications $P_i \Rightarrow F_i$ are referred

[3] This comes from Definition 15 in [2]

to as *rules*. Since neither LTL nor CTL have explicit past-time operators, Bolotov and Fisher [3] define the **start** operator which holds only at the beginning of time.

$$(M, \pi) \models_k^i \mathbf{start} \Leftrightarrow \pi(i) \in I$$

The possible rules are thus

$\mathbf{start} \Rightarrow \bigvee_j l_j$ An *initial* rule $\quad \bigwedge_i l_i \Rightarrow \mathbf{X} \bigvee_j l_j$ A *global* **X**-rule

$\bigwedge_i l_i \Rightarrow \mathbf{F} \bigvee_j l_j$ A *global* **F**-rule

where l_i and l_j are literals.

The transformation functions $T(\Psi)$ recursively convert a set of rules which do not conform to the normal form into a set of rules which do. To convert any temporal logic formula f to SNF, it is sufficient to apply the transformation rules to the singleton set $\{\mathbf{start} \Rightarrow f\}$. For brevity, we do not list the full set of transformations here; in general they are trivially adapted from those in [3], or from standard propositional logic.

$$T_{\mathrm{G}}(\{P \Rightarrow \mathbf{G}\, f\} \cup \Psi) = \left\{ \begin{array}{c} P \Rightarrow f \wedge x \\ x \Rightarrow \mathbf{X}\,(f \wedge x) \end{array} \right\} \cup T_{\mathrm{G}}(\Psi)$$

$$T_{\mathrm{U}}(\{P \Rightarrow f\, \mathbf{U}\, g\} \cup \Psi) = \left\{ \begin{array}{c} P \Rightarrow g \vee (f \wedge x) \\ x \Rightarrow \mathbf{X}\,(g \vee (f \wedge x)) \\ P \Rightarrow \mathbf{F}\, g \end{array} \right\} \cup T_{\mathrm{U}}(\Psi)$$

$$T_{\mathrm{ren1}}(\{P \Rightarrow \mathbf{G}\, f(\mathbf{F}\, g)\} \cup \Psi) = \left\{ \begin{array}{c} P \Rightarrow \mathbf{G}\, f(x) \\ x \Rightarrow \mathbf{F}\, g \end{array} \right\} \cup T_{\mathrm{ren1}}(\Psi)$$

In each of the above transformations, a new variable x is introduced: the conversion to SNF introduces one variable for each removed operator (in the first two transformations above) in addition to the renaming variables used to flatten the structure of the formula (in the last transformation above).

The transformations to rules are based on the fixpoint characterisations of the LTL operators. All LTL operators can be represented as the fixpoint of a recursive function [8]; the transformations encode the corresponding function as a rule which is required to hold in all states. Only those operators characterised by greatest fixpoints are converted (*always* (**G**) and *weak until* (**W**); *until* (**U**) is first converted to *weak until* and *sometime* for its transformation) which means that the *sometime* operator remains unchanged.

By Tarski's fixpoint theorem [18] we know that a finite number of iterations of a rule is sufficient to find its fixpoint. Thus the instance of the introduced variable at time i holds iff the original operator held at time i. For a formal proof of the correctness of the transformations, see [10].

3.2 Bounded SNF

Although the fixpoint characterisations are given for *unbounded* temporal logic, they are preserved for most of bounded LTL since we have bounded semantics for *next-state* ($\mathbf{X}$). We note that the characterisation of *always* is valid if and only if the path is a k-loop; we encapsulate this constraint in the new operator *next-loop-state* ($\mathbf{X_l}$) with semantics

$$(M,\pi) \models_k^i \mathbf{X_l}\, f_1 \Leftrightarrow \begin{cases} (M,\pi(i+1)) \models_k f_1 & \text{if } \pi \text{ is a } k\text{-loop} \\ \bot & \text{otherwise} \end{cases}$$

and modify the transformation accordingly.

The bounded semantics of *always* also fails to capture the concept of rules holding in *all reachable states*. We give the semantics for a modified operator *bounded always* ($\mathbf{G_k}$) for bounded LTL without the restriction to paths with loops.

$$(M,\pi) \models_k^i \mathbf{G_k}\, f_1 \Leftrightarrow \begin{cases} \forall j, i \leq j.(M,\pi(j)) \models_k f_1 & \text{if } \pi \text{ is a } k\text{-loop} \\ \forall j, i \leq j < k.(M,\pi(j)) \models_k f_1 & \text{otherwise} \end{cases}$$

The correctness of the transformations rely on a sufficient number of instances of the rules occurring. In BMC, this means that the transformations based on fixpoints are correct only when the bound is sufficiently large. It is easy to see, by appealing to the semantics, that the failure mode with an insufficiently large bound is the same as that for the original encoding: no counterexample is found.

Introducing this operator allows us to restate the general form as

$$\mathbf{G_k}\left(\bigwedge_i (P_i \Rightarrow F_i)\right)$$

The rules $P_i \Rightarrow F_i$ are now of the following form:

$$\mathbf{start} \Rightarrow \bigvee_j l_j \quad \text{An } \textit{initial} \text{ rule} \qquad \bigwedge_i l_i \Rightarrow \mathbf{X_l} \bigvee_j l_j \quad \text{A } \textit{global } \mathbf{X_l}\text{-rule}$$

$$\bigwedge_i l_i \Rightarrow \mathbf{X} \bigvee_j l_j \quad \text{A } \textit{global } \mathbf{X}\text{-rule} \qquad \bigwedge_i l_i \Rightarrow \mathbf{F} \bigvee_j l_j \quad \text{A } \textit{global } \mathbf{F}\text{-rule}$$

with the transformation for the *always* operator being amended to

$$T_{\mathrm{G}}(\{P \Rightarrow \mathbf{G}\, f\} \cup \Psi) = \left\{ \begin{array}{c} P \Rightarrow f \wedge x \\ x \Rightarrow \mathbf{X_l}(f \wedge x) \end{array} \right\} \cup T_{\mathrm{G}}(\Psi)$$

The correctness of bounded SNF is covered in [16].

Table 2. The BMC Encoding for SNF-LTL

f	$[\![f]\!]^0_k$	${}_l[\![f]\!]^0_k$
$\mathbf{start} \Rightarrow f_1$	$[\![f_1]\!]^0_k$	${}_l[\![f_1]\!]^0_k$
$\mathbf{G_k}(f_1 \Rightarrow \mathbf{X_l}\, f_2)$	$\bot$	$\bigwedge_{n=0}^{k} \left({}_l[\![f_1]\!]^n_k \Rightarrow {}_l[\![f_2]\!]^{n+1}_k\right)$
$\mathbf{G_k}(f_1 \Rightarrow \mathbf{X}\, f_2)$	$\bigwedge_{n=0}^{k-1} \left([\![f_1]\!]^n_k \Rightarrow [\![f_2]\!]^{n+1}_k\right)$	$\bigwedge_{n=0}^{k} \left({}_l[\![f_1]\!]^n_k \Rightarrow {}_l[\![f_2]\!]^{n+1}_k\right)$
$\mathbf{G_k}(f_1 \Rightarrow \mathbf{F}\, f_2)$	$\bigwedge_{n=0}^{k} \left([\![f_1]\!]^n_k \Rightarrow \bigvee_{m=n}^{k} [\![f_2]\!]^m_k\right)$	$\bigwedge_{n=0}^{k} \left({}_l[\![f_1]\!]^n_k \Rightarrow \bigvee_{m=\min(n,l)}^{k} {}_l[\![f_2]\!]^m_k\right)$

3.3 Encoding Bounded SNF

The distributivity of *bounded always* follows directly from its semantics; because of the unusual semantics of **start**, this means that any LTL formula may be represented as a conjunction of instances of the following 'universal' rules:

$$\mathbf{start} \Rightarrow \bigvee_j l_j \qquad \mathbf{G_k}\left(\bigwedge_i l_i \Rightarrow \mathbf{X_l} \bigvee_j l_j\right)$$

$$\mathbf{G_k}\left(\bigwedge_i l_i \Rightarrow \mathbf{X} \bigvee_j l_j\right) \qquad \mathbf{G_k}\left(\bigwedge_i l_i \Rightarrow \mathbf{F} \bigvee_j l_j\right)$$

Although it is simple to encode these rules using the standard BMC encodings in Table 1, we can take advantage of the limited nesting depth characteristic of these normal forms to define a more efficient encoding, in the same way as for the depth 1 case in [4] and [17]. We give the more efficient encodings in Table 2. Note that although we make use of the BMC encodings, they are only used for purely propositional formulæ. No further proof of these encodings is required: they are trivial simplifications of those proved in [2].

For propositional f, $[\![f]\!]^i_k \equiv {}_l[\![f]\!]^i_k$, so we can deduce from Table 2 that this relationship also holds for many cases where f is a rule. Under these circumstances, we can factorise the encodings for f out of the disjunctions in Equation 1 either explicitly during the encoding or by processing the resulting propositional formula. Often the checks for the looping nature of π will cancel each other out entirely, further simplifying the encoding. While this type of optimisation can be made with the standard BMC encoding, it only occurs where operators are not nested; the renaming effect of SNF simplifies this optimisation and makes it more widely applicable.

3.4 The Fixpoint Normal Form

We noted in Section 3.1 that SNF converts only the greatest fixpoint operators, leaving rules containing the *sometime* operator; we see from Table 2 that these rules are the pathological case for this encoding. Converting the *sometime* operator in the same way requires care.

A transformation based directly on the fixpoint characterisation would be

$$T_F(\{P \Rightarrow \mathbf{F}\, f\} \cup \Psi) = \left\{ \begin{array}{c} P \Rightarrow f \vee x \\ x \Rightarrow \mathbf{X}\,(f \vee x) \end{array} \right\} \cup T_{\mathrm{F}}(\Psi)$$

The problem stems from the disjunction in the second rule. Since we are trying to show satisfiability, it is simple to satisfy each occurrence of the rule by setting the right hand disjunct to true for all time: the rule can always be satisfied. Since we are interested only in the bounded semantics of the operator, it is possible to break this chain at the bound by introducing an extra operator:

$$(M, \pi) \models^i_k \mathbf{bound} \Leftrightarrow i \geq k$$

The transformation is now

$$T_F(\{P \Rightarrow \mathbf{F}\, f\} \cup \Psi) = \left\{ \begin{array}{c} P \Rightarrow f \vee x \\ x \Rightarrow \mathbf{X}\,(f \vee x) \\ \mathbf{bound} \Rightarrow f \vee \neg x \end{array} \right\} \cup \Psi$$

3.5 Correctness of the Fixpoint Normal Form Transformation

We take the outline of the proof from [10]. For a transformation T to preserve the semantics of an arbitrary formula f, we require that

> for all models M and for all LTL formulæ f, $(M, \pi) \models_k f$ iff there exists an M' such that $M \sim^x M'$ and $(M, \pi) \models_k \tau(f)$

where x is a new propositional variable introduced, and $M \sim^x M'$ if and only if M differs from M' in at most the valuation given x. We express this in temporal logic with quantification over propositions (QPTL)[4] as $\vdash_{\mathrm{QPTL}} f \Leftrightarrow \exists x.T(f)$. The proof is given for the case that the rule set is a singleton set, since for all transformations, T is independent of Ψ. The proofs may easily be extended to non-empty Ψ.

Lemma 1. *For sufficiently large k, $(M, \pi) \models_k \mathbf{F}\, f_1$ if and only if $(M', \pi) \models_k (x \vee f_1)$ and $(M', \pi) \models_k \mathbf{G_k}(x \Leftrightarrow \mathbf{X}(x \vee f_1))$ where $M \sim^x M'$.*

Proof. Consider the fixpoint expression $\tau(Z) = f_1 \vee \mathbf{X}\, Z$. We introduce the variable x such that for all n,

$$(M', \pi) \models^n_k x \Leftrightarrow (M', \pi) \models^n_k \mathbf{X}\, \tau^{k-n}(\mathsf{true})$$

By substituting the definition of x and by one substitution of the definition of τ, we have $(M', \pi) \models^n_k x \Leftrightarrow (M', \pi) \models^n_k \mathbf{X}\,(f_1 \vee x)$ and by reference to the semantics, $(M', \pi) \models_k \mathbf{G_k}(x \Leftrightarrow \mathbf{X}(x \vee f_1))$.

From the least fixpoint characterisation[8], $(M', \pi) \models_k x \Leftrightarrow \mathbf{F}\, f_1$, and by unrolling τ by one step and substituting the definition of x, we get $(M', \pi) \models_k f_1 \vee x$.

[4] See [19] for full details; briefly, $(M, i) \models \exists p.A$ iff there exists an M' such that $(M', i) \models A$, and M' and M differ at most in the valuation given to p.

Theorem 1. *For any rule* A, $\vdash_{\mathrm{QPTL}} A \Leftrightarrow \exists x.T_F(A)$

Proof. Proving each direction independently:

- $\vdash_{\mathrm{QPTL}} A \Rightarrow \exists x.T_F(A)$
  Substituting Lemma 1,

$$\begin{aligned}
&\mathbf{G_k}(P \Rightarrow \mathbf{F}\, B) \\
&\quad \Rightarrow \exists x.\mathbf{G_k}(x \Leftrightarrow \mathbf{X}(x \vee B)) \wedge \mathbf{G_k}(\mathbf{bound} \Rightarrow \neg x) \wedge \mathbf{G_k}(P \Rightarrow (x \vee B)) \\
&\quad \Rightarrow \exists x.\mathbf{G_k}((x \Leftrightarrow \mathbf{X}(x \vee B)) \wedge \mathbf{bound} \Rightarrow \neg x \wedge (P \Rightarrow (x \vee B)))
\end{aligned}$$

  which implies the set of rules $\{x \Rightarrow \mathbf{X}(x \vee B), \mathbf{bound} \Rightarrow \neg x, P \Rightarrow x \vee B\}$.
- $\vdash_{\mathrm{QPTL}} \exists x.T_F(f) \Rightarrow f$
  Starting with the transformed set of rules $\{x \Rightarrow \mathbf{X}(x \vee B), \mathbf{bound} \Rightarrow \neg x, P \Rightarrow x \vee B\}$, and exploiting the corollary of Lemma 1, $(M', s_i) \models_k (x \vee f_1) \Leftrightarrow (M', s_i) \models_k \mathbf{F}\, f_1$ iff $(M', s_i) \models \mathbf{G_k}(\mathbf{bound} \Rightarrow \neg x)$

$$\begin{aligned}
&\mathbf{G_k}((x \Leftrightarrow \mathbf{X}(x \vee B)) \wedge \mathbf{bound} \Rightarrow \neg x \wedge (P \Rightarrow (x \vee B))) \\
\Leftrightarrow\ &\mathbf{G_k}(x \Leftrightarrow \mathbf{X}(x \vee B)) \wedge \mathbf{G_k}(\mathbf{bound} \Rightarrow \neg x) \wedge \mathbf{G_k}(P \Rightarrow (x \vee B)) \\
\Leftrightarrow\ &\mathbf{G_k}(x \Leftrightarrow \mathbf{X}\,\mathbf{F}\, B) \wedge \mathbf{G_k}(\mathbf{bound} \Rightarrow \neg x) \wedge \mathbf{G_k}(P \Rightarrow (x \vee B)) \\
\Rightarrow\ &\mathbf{G_k}((x \Rightarrow \mathbf{X}\,\mathbf{F}\, B) \wedge (P \Rightarrow (x \vee B))) \\
\Rightarrow\ &\mathbf{G_k}(P \Rightarrow ((\mathbf{X}\,\mathbf{F}\, B) \vee B)) \\
\Rightarrow\ &\mathbf{G_k}(P \Rightarrow \mathbf{F}\, B)
\end{aligned}$$

  That is, the singleton rule set $\{P \Rightarrow \mathbf{F}\, B\}$.

4 Comparisons

We compare the encodings on an example specification $\mathbf{G}\,\mathbf{F}\, f$. This is a reachability specification, with many applications. Before encoding, the specification is negated to

$$\mathbf{F}\,\mathbf{G}\,\neg f \tag{2}$$

We consider only the loop encoding, as the non-loop encoding is $\perp$ for all methods due to the semantics of *always*.

The original, recursive encoding decomposes in two steps.In the loop case,

$$\begin{aligned}
{}_l[\![\mathbf{F}\,\mathbf{G}\,\neg f, \pi]\!]_k^0 &= \bigvee_{i=0}^{k} {}_l[\![\mathbf{G}\,\neg f, \pi]\!]_k^i \\
&= \bigvee_{i=0}^{k} \bigwedge_{j=\min(i,l)}^{k} f(j)
\end{aligned}$$

This is a disjunction of conjunctions: the pathological case for conversion to clause form. It is possible to define a more efficient encoding using renamed

subformulæ [4], but this approach is difficult to generalise. The size of the formula is $O(k^2)$, hence the cost to build it before CNF conversion is quadratic.

The conversion to SNF yields the following rules[5]

$$\begin{aligned} \mathbf{start} &\Rightarrow \mathbf{F}\, x_1 \\ x_1 &\Rightarrow \neg f \wedge x_2 \\ x_2 &\Rightarrow \mathbf{X_l}\,(\neg f \wedge x_2) \end{aligned}$$

which encode to the three conjuncts

$$\begin{aligned} &\bigvee_{i=0}^{k} x_1(i) \quad \wedge \\ &\bigwedge_{i=0}^{k} (x_1(i) \Rightarrow \neg f(i) \wedge x_2(i)) \quad \wedge \\ &\bigwedge_{i=0}^{k} (x_2(i) \Rightarrow \neg f(i+1) \wedge x_2(i+1)) \end{aligned}$$

We have two introduced variables: the first establishes a renaming of the $\mathbf{G}\,\neg f$ subformula, and the second renames each successive step of this subformula. This means that steps are shared between references from the $\mathbf{F}$ operator, leading to a simplification of the problem which is easier to solve as well as being smaller. The added complexity of the introduced variables is balanced by the ability to reuse subformulæ many times.

The encoding corresponds to an ideal renaming of the formula above, but the conversion is performed in linear time, and results in a formula of size $O(k)$. Furthermore, we can show in advance that the encoding of each rule used here is invariant with respect to l, which means that the subformulæ can be factorised out of the disjunction of loops seen in Equation 1.

Finally, we examine the fixpoint normal form conversion. The set of rules corresponding to the specification is

$$\begin{aligned} \mathbf{start} &\Rightarrow x_0 \vee x_1 \\ x_0 &\Rightarrow \mathbf{X}\,(x_0 \vee x_1) \\ \mathbf{bound} &\Rightarrow x_1 \vee \neg x_0 \\ x_1 &\Rightarrow \neg f \wedge x_2 \\ x_2 &\Rightarrow \mathbf{X_l}\,(\neg f \wedge x_2) \end{aligned}$$

[5] Further reduction of the second and third rules is necessary for correct SNF; we disregard this as it makes no difference to the final encoding

which encode to the conjuncts

$$\begin{aligned} & x_0(0) \vee x_1(0) \quad \wedge \\ & \bigwedge_{i=0}^{k} (x_0(i) \Rightarrow x_0(i+1) \vee x_1(i+1)) \quad \wedge \\ & x_1(k) \vee \neg x_0(k) \quad \wedge \\ & \bigwedge_{i=0}^{k} (x_1(i) \Rightarrow \neg f(i) \wedge x_2(i)) \quad \wedge \\ & \bigwedge_{i=0}^{k} (x_2(i) \Rightarrow \neg f(i+1) \wedge x_2(i+1)) \end{aligned}$$

The main difference between the SNF encoding and the fixpoint normal form encoding is the omission of the long disjunction in the first conjunct which would be encoded as a single long clause. This is replaced by an array of conjunctions which rename each step in much the same way as for the **G** operator. Although in this case the advantage is dependent on the SAT checker, it is clear that where the **F** operator is nested, similar advantages would be seen as for SNF with the **G** operator.

5 Results

We compare the SNF and Fixpoint encodings with the encoding used in NuSMV version 2.0.2; this version of NuSMV includes several of the optimisations discussed in [4]. For consistency, we have implemented the SNF and Fixpoint encodings as options in NuSMV. All of the experiments have been done using the SAT solver zChaff [15] on a 700MHz AMD Athlon with 256Mb main memory, running Linux.

5.1 Scalability

We observe the difference in the behaviour of the encodings with increasing problem size by choosing a simple problem that is easy to scale. The benchmark circuits have been kept deliberately simple as it is the encoding of the specification not the model that differentiates the encodings.

A shift register is a storage device of length n which, at each time step, moves the values stored in each of its elements to the next element, reading in a new value to fill the now empty first element. That is, storage elements $x_0 \ldots x_{n-1}$ and input in are transformed such that $\forall i, 0 < i < n \cdot (x_i \leftarrow x_{i-1})$ and $x_0 \leftarrow in$. The specification that the shift register must fulfil will depend on its application; we explore a number of response patterns taken from [6]. The specifications depend on the number of elements in the shift register, referring to points at the end and middle of the register. For example, in the case of a three element register:

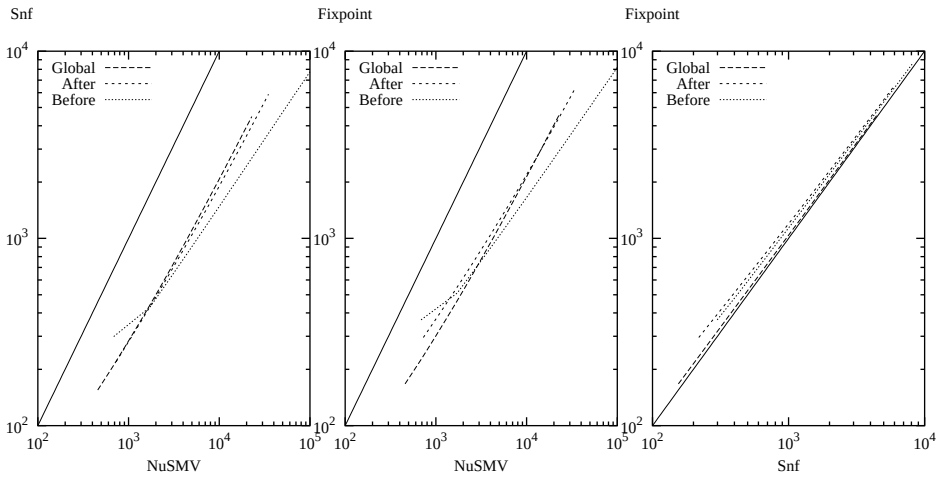


Fig. 2. Number of clauses generated by a shift register model

- Global response (depth 2) — x_2 goes high in response to in: $\mathbf{G}(in \Rightarrow \mathbf{F}\, x_2)$
- After response (depth 3)— x_2 goes high in response to in, after x_1 has gone high: $\mathbf{G}\,(x_1 \Rightarrow \mathbf{G}(in \Rightarrow \mathbf{F}\, x_2))$
- Before response (depth 3)— x_1 goes high in response to in, before x_2 has gone high (this property is only true if all the registers are zero, so we test for $empty \equiv \neg x_0 \wedge \neg x_1 \wedge \neg x_2$ too): $[((in \wedge empty) \Rightarrow [\neg x_2\, \mathbf{U}\, (x_1 \wedge \neg x_2)])\, \mathbf{U}\, (x_2 \vee \mathbf{G}\, x_2)]$

Number of Clauses. We see in Figure 2 that the number of clauses produced by both SNF and Fixpoint grows, in general, less quickly than the number produced by NuSMV, as the length of the register increases. The differing gradients follow the behaviour predicted by the differing depths of the specifications: the slopes become shallower with increasing depth indicating an exponential improvement in the number of clauses.

The advantage of the Fixpoint encoding over SNF is dependent upon the number of occurrences of the *always* operator in the specification, since this is the only difference between the encodings. We see the greatest advantage for Fixpoint in the after response and before response specifications, with two occurrences of the *always* operator; the first operator in the after response specification has a smaller encoding than the second as one of the corresponding rules is an initial rule.

We can conclude that, as far as the number of clauses is concerned, the Fixpoint encoding outperforms SNF and NuSMV in the way that is expected: size and rate of size increase decreasing with the nesting depth and the occurrence of least fixpoint operators.

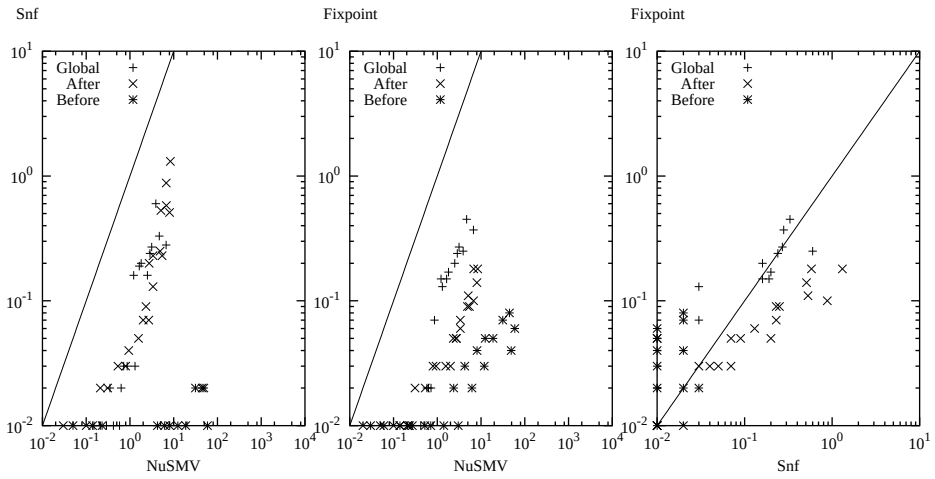


Fig. 3. Time taken by zChaff for a shift register model

zChaff timings. Counting the number of clauses is far from being an effective method of determining the efficiency of an encoding. We also look at one of the current state-of-the-art SAT solvers, zChaff [15].

The behaviour is far less clear than for the number of clauses; zChaff is a complex system. Broadly, the SNF and Fixpoint encodings always result in a shorter runtime than the NuSMV encoding; the Fixpoint encoding outperforms the SNF encoding only for the after response specification (for the global response specification, the trend is towards an improvement for larger problems).

We see a clear exponential improvement for certain specifications: the timings for Before with SNF and Fixpoint grow exponentially slower than NuSMV; the global response specification shows the same trend less dramatically. We only see a exponential improvement for the after response specification with the Fixpoint encoding: with the SNF encoding, the trend appears to be towards NuSMV being faster.

5.2 Distributed Mutual Exclusion

The distributed mutual exclusion circuit from [13] forms a good basis for comparing the performance of different encodings as it meaningfully implements several specifications. We look at three here, applied to a DME of four elements:

- Accessibility: if an element wishes to enter the critical region, it eventually will. We check the accessibility of the first two elements. This specification is correct, so as in [2], we check at a chosen bound to illustrate the timing differences.
  $\mathbf{G}(\mathit{request}(0) \rightarrow \mathbf{F}\ \mathit{enter}(0)) \wedge \mathbf{G}(\mathit{request}(1) \rightarrow \mathbf{F}\ \mathit{enter}(1))$
- Precedence given token possession: the mutual exclusion property is enforced by a token passing mechanism; if an element of the DME holds the token,

Table 3. Timing results in zChaff for the distributed mutual exclusion circuit

Specification	Bound	NuSMV encoding	SNF encoding	Fixpoint encoding	SMV
Accessibility	30	2.65	0.33	0.36	13.13
Accessibility	40	20.93	4.84	4.33	13.13
Priority for 0	14	0.13	0.02	0.02	12.97
Priority for 1	54	14.93	0.44	0.76	15.00
Overtaking depth 1	40	85.73	2.15	1.11	13.96
Overtaking depth 2	40	*	4.92	5.15	14.14

then its requests to enter the critical region are given precedence. We check the converse: if the first element holds the token, the second does not have precedence and *vice versa*. Since the token begins at the first element, this is the quicker to prove, with a bound of 14. For the second element, a bound of 54 is required to find the counterexample.
$\mathbf{G}((request(0) \wedge request(1) \wedge token(0)) \rightarrow [\neg enter(0)\, \mathbf{U}\, enter(1)])$

– Bounded overtaking given token possession: if two elements wish to enter the critical region, then the higher priority may enter a given number of times before the other. We check bounded overtaking of one and two entrances. Both specifications are correct so as above we check at a bound of 40. These specifications are the most complex, including up to four nested *until* operators.
For one entrance: $\mathbf{G}((request(0) \wedge request(1) \wedge token(0)) \rightarrow [(\neg enter(0) \wedge \neg enter(1))\, \mathbf{U}\, (enter(0) \wedge \mathbf{X}(enter(0)\, \mathbf{U}\, [(\neg enter(0) \wedge \neg enter(1))\, \mathbf{U}\, enter(1)]))])$

The results are summarised in Table 3 together with the timings for CMU SMV on CTL representations of the same problems[6]. For the bounded overtaking problems, we note that NuSMV took nearly 10 minutes to generate the formula in the first case, and after 25 minutes had not completed in the second case. In contrast, the time taken to perform the SNF and Fixpoint encodings were insignificant.

While both the SNF and Fixpoint encodings outperform the NuSMV encoding and SMV, we do not see a consistent advantage to either. The results for accessibility suggest that Fixpoint scales better with increasing bound, while the results for bounded overtaking suggest that SNF scales better with increasing specification depth.

5.3 Texas-97 Benchmarks

We examine a number of model checking benchmarks from the *Texas-97* benchmark suite [1]. These benchmarks have been converted from the Blif-mv representation to SMV format by a locally modified version of the VIS model checker [12]. We run these benchmarks at fixed bounds and report the time spent by zChaff.

[6] We note that for SMV to terminate in a reasonable time on these problems, it must be started with the `-inc` switch. No similar knowledge of model checker behaviour is needed for BMC.

Table 4. Timing results in zChaff for the MSI cache coherence protocol

Processors	Specification	Bound	NuSMV	SNF	Fixpoint
2	Request A	10	4.40	1.73	1.53
2	Request A	20	19.40	5.82	9.97
2	Request B	10	2.65	3.63	2.69
2	Request B	20	49.78	8.63	16.42
3	Request A	10	13.00	3.03	2.50
3	Request A	20	39.22	8.2	5.79
3	Request B	10	4.60	6.66	5.93
3	Request B	20	54.94	62.11	40.25
3	Request C	10	4.58	6.64	5.91
3	Request C	20	44.8	50.27	37.65

MSI Cache Coherence Protocol This is an implementation of a Modified Shared Invalid cache coherence protocol between two or three processors. We examine two of the specifications of behaviour from the benchmark. The results are summarised in Table 4.

- Whenever processor A requests the bus, it gets the bus in the next clock cycle. Listed as "Request A" in the results. $\mathbf{G}(\mathit{bus\_reqA} \rightarrow \mathbf{X}\ \mathit{bus\_ackA})$
- Whenever processor B (or C) requests the bus, it gets the bus only when Processor A did not request the bus. Listed as "Request B" or "Request C" in the results. $\mathbf{G}(\mathit{bus\_reqB} \rightarrow \mathbf{F}\ \mathit{bus\_ackB})$

Instruction Fetch Control Module This is a model of the instruction fetch control module of the experimental TORCH microprocessor developed at Stanford University. Three models are examined; from the text accompanying the benchmark set:

- IFetchControl1: The original instruction module with several assumptions on the environmental signal.
- IFetchControl2: As IFetchControl1 except that the memory stall signal is always low.
- IFetchControl3.v: As IFetchControl1 except that the instruction cache line is assumed to be always valid.

We examine three specifications from the benchmark. The results are summarised in Table 5.

- The delayed version of a signal should, in the next state, have the signal's previous value. Listed as "Delay" in the results. $\mathbf{G}(\mathit{IStall\_s1} \rightarrow \mathbf{X}\ \mathit{IStall\_s2})$
- As above, for the Refetch state. Listed as "Refetch" in the results. $\mathbf{G}((\mathit{PresState\_s1} = \mathit{REFETCH}) \rightarrow \mathbf{X}((\mathit{PrevState\_s2} = \mathit{REFETCH}))$
- $\mathit{WriteCache\_s2}$ becomes one in some paths before $\mathit{WriteTag\_s2}$ becomes one. Listed as "WriteCache" in the results. $\neg[\neg\, \mathit{WriteTag\_s2}\ \mathbf{U}\ (\mathit{WriteCache\_s2} \wedge \neg\, \mathit{WriteTag\_s2})]$

Table 5. Timing results in zChaff for the Instruction Fetch Control Module

Model	Specification	Bound	NuSMV	SNF	Fixpoint
IFetchControl1	Delay	10	0.94	0.45	0.44
IFetchControl2	Delay	10	0.99	0.40	0.40
IFetchControl3	Delay	10	1.29	0.39	0.50
IFetchControl1	Refetch	10	3.69	0.91	0.82
IFetchControl2	Refetch	10	3.30	0.89	0.81
IFetchControl3	Refetch	10	3.74	1.49	1.88
IFetchControl1	WriteCache	10	3.58	1.68	2.47
IFetchControl2	WriteCache	10	2.67	1.65	1.78
IFetchControl3	WriteCache	10	2.78	2.24	1.40

Table 6. Timing results in zChaff for the Pentium Pro Split-Transaction Bus

Opcode 0	Opcode 1	Specification	NuSMV	SNF	Fixpoint
Load2Store	Load2Store	IOQ	949.53	202.83	202.90
Load2Store	Store	IOQ	753.26	156.43	156.24
Load2Store	Load2Store	Live 1	923.12	176.64	169.97
Load2Store	Load	Live 1	745.94	131.61	145.88
Load2Store	Store	Live 1	1111.63	175.58	199.19
Load2Store	Load2Store	Live 2	919.61	167.23	160.38
Load2Store	Load	Live 2	883.51	134.52	155.23
Load2Store	Store	Live 2	738.74	128.96	143.04

Pentium Pro Split-Transaction Bus This is a model of the Modified Exclusive Shared Invalid cache coherence protocol used by the Intel Pentium Pro processor for SMP. We examine a number of different combinations of opcodes running on the processors, with the memory address of the transaction being nondeterministically 0 or 1.

We examine three specifications from the benchmark. The results are summarised in Table 6.

- Correctness of the bus transaction IOQ. Listed as "IOQ" in the results.
  $\mathbf{G}(\neg((processor0.fifo = REQUEST) \wedge (processor1.fifo = REQUEST)))$
- Liveness of processor 0 (part 1). Listed as "Live 1" in the results.
  $\mathbf{G}((processor0.stage = FETCH) \rightarrow \mathbf{F}(processor0.stage = EXECUTE))$
- Liveness of processor 0 (part 2). Listed as "Live 2" in the results.
  $\mathbf{G}((processor0.stage = EXECUTE) \rightarrow \mathbf{F}(processor0.stage = FETCH))$

Summary While we can see that the SNF and Fixpoint encodings outperform the NuSMV encoding in many cases, the gains are typically less dramatic than were seen for the mutual exclusion circuit. The models are encoded in the same way regardless of the encoding used for the specification; these benchmarks are very large circuits with several thousand variables, so it is reasonable to suppose that the performance gains due to the new encodings are mitigated by the time taken to process the model.

The specifications used in these benchmarks are much simpler than those used to test the DME: typically of the form $\mathbf{G}(a \rightarrow \mathbf{X}\, b)$ or $\mathbf{G}(a \rightarrow \mathbf{F}\, b)$. This suggests again that one advantage of the SNF and Fixpoint encodings are dependent on the nesting depth of the specification.

6 Conclusions

We have described two new encoding schemes for bounded model checking which build on the existing encodings and use the fixpoint characterisations of LTL. The first is a novel application of the Separated Normal Form, while the second extends SNF by the introduction of a transformation for the *eventually* operator. We have shown that these new encodings are correct, provided that the original bounded model checking encoding is correct.

We have demonstrated a reduction in the number of clauses generated by the problem which is exponential in the size of the problem instance, for both encodings, and also that the improvement in performance in the SAT checker can be exponential in the size of the problem instance, depending on the specification. We have demonstrated a clear performance advantage to these encodings over the NuSMV bounded model checking implementation in several real-world examples, and we have demonstrated the advantage that these encodings give BMC over conventional symbolic model checkers.

References

1. Adnan Aziz et al. Examples of HW verification using VIS, 1997. `http://vlsi.colorado.edu/~vis/texas-97/`
2. Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In W.R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems. 5th International Conference, TACAS'99*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag Inc., July 1999.
3. Alexander Bolotov and Michael Fisher. A resolution method for CTL branching-time temporal logic. In *Proceedings of the Fourth International Workshop on Temporal Representation and Reasoning (TIME)*. IEEE Press, 1997.
4. Alessandro Cimatti, Marco Pistore, Marco Roveri, and Roberto Sebastiani. Improving the encoding of LTL model checking into SAT. In Agostino Cortesi, editor, *Third International Workshop on Verification, Model Checking and Abstract Interpretation*, volume 2294 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., January 2002.
5. Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.
6. M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Property Specification Patterns for Finite-State Verification. In M. Ardis, editor, *2nd Workshop on Formal Methods in Software Practice*, pages 7–15, March 1998.
7. M.B. Dwyer, G.S. Avruning, and J.C. Corbett. Patterns in property specifications for finite-state verification. In *21st International Conference on Software Engineering, Los Angeles, California*, May 1999.

8. E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In Jan van Leeuwen J. W. de Bakker, editor, *Automata, Languages and Programming, 7th Colloquium*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer-Verlag Inc, 1980.
9. Michael Fisher. A resolution method for temporal logic. In *Proceedings of Twelfth International Joint Conference on Artificial Intelligence (IJCAI)*. Morgan Kaufmann, August 1991.
10. Michael Fisher and Philippe Noël. Transformation and synthesis in METATEM Part I: Propositional METATEM. Technical Report UMCS-92-2-1, Department of Computer Science, University of Manchester, Manchester M13 9PL, England, February 1992.
11. Dov Gabbay. The declarative past and imperative future. In H. Barringer, editor, *Proccedings of the Colloquium on Temporal Logic and Specifications*, volume 398 of *Lecture Notes in Computer Science*, pages 409–448. Springer-Verlag, 1989.
12. The VIS Group. VIS: A system for verification and synthesis. In R. Alur and T. Henzinger, editors, *Proceedings of the 8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432, New Brunswick, NJ, July 1996. Springer.
13. A. J. Martin. The design of a self-timed circuit for distributed mutual exclusion. In Henry Fuchs, editor, *Proceedings of the 1985 Chapel Hill Conference on VLSI*, pages 245–260. Computer Science Press, 1985.
14. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, 1992.
15. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *39th Design Automation Conference*, Las Vegas, June 2001.
16. Daniel Sheridan. Using fixpoint characterisations of LTL for bounded model checking. Technical Report APES-41-2002, APES Research Group, January 2002. Available from `http://www.dcs.st-and.ac.uk/~apes/apesreports.html`
17. Daniel Sheridan and Toby Walsh. Clause forms generated by bounded model checking. In Andrei Voronkov, editor, *Eighth Workshop on Automated Reasoning*, 2001.
18. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.
19. Pierre Wolper. Specification and synthesis of communicating processes using an extended temporal logic. In *Proceeding of the 9th Symposium on Principles of Programming Languages*, pages 20–33, Albuquerque, January 1982.

Bounded Verification of Past LTL*

Alessandro Cimatti[1], Marco Roveri[1], Daniel Sheridan[2]

[1] Istituto per la Ricerca Scientifica e Tecnologica (IRST)
Via Sommarive 18, 38050 Povo, Trento, Italy
{cimatti,roveri}@irst.itc.it
[2] School of Informatics, The University of Edinburgh
Kings Buildings, Mayfield Road, Edinburgh EH9 3JZ, UK
d.j.sheridan@sms.ed.ac.uk

Abstract. Temporal logics with past operators are gaining increasing importance in several areas of formal verification for their ability to concisely express useful properties. In this paper we propose a new approach to bounded verification of PLTL, the linear time temporal logic extended with past temporal operators. Our approach is based on the transformation of PLTL into Separated Normal Form, which in turn is amenable for reduction to propositional satisfiability. An experimental evaluation shows that our approach induces encodings which are significantly smaller and more easily solved than previous approaches, in the cases of both model checking and satisfiability problems.

1 Introduction

Temporal logics with past operators are being devoted increasing interest in a number of application areas (e.g. formal verification [12, 5, 10], requirement engineering [13, 17], and automated task planning [2]). In the widely-used setting of Linear Temporal Logics (LTL), past operators do not add expressive power with respect to pure-future: any LTL formula with past operators can be rewritten by only using future-time operators [11]. On the other hand, past operators are very useful in practice, since they help to keep specifications compact, simple, and easy to understand. This practical consideration has a formal counterpart in the fact that LTL with past operators is exponentially more succinct than LTL with pure-future operators [14].

In this paper we tackle the problem of lifting SAT-based verification techniques, which are becoming a prominent technology in many application areas, to deal with past operators. We focus on *bounded* verification for PLTL, where the analysis is limited to behaviors of a fixed number of time steps.

* This work is partially sponsored by the PROSYD EC project, contract number IST-2003-507219, and the CALCULEMUS! IHP-RTN EC project, contract code HPRN-CT-2000-00102, and has thus benefited of the financial contribution of the Commission through the IHP programme. We thank Paul Jackson, Roberto Sebastiani and Simone Semprini for their useful comments and feedback.

Our interpretation of bounded verification encompasses both Bounded Model Checking and Bounded Satisfiability. *Bounded Model Checking* [4] focuses on design verification: given a model M (typically representing a design) and a formula φ (typically representing a desired property), checking the existence of a counterexample (a behavior of M which violates φ) over k steps is reduced to a purely propositional satisfiability problem, and solved with an efficient SAT solver.

Bounded Satisfiability is more directed to the analysis of requirements, which is gaining a significant practical interest. In fact, we are witnessing the take off of property-based design paradigms (e.g. with the acceptance of the PSL/Sugar language [1] as a IEEE standard language for property specification). This highlights the increased recognition of the importance of properties that are intended to specify the design intent, rather than the design itself. The object of the verification is now a set of requirements, represented as a set of PLTL formulae Γ. Different forms of analysis can be envisaged: for instance, we may be interested in checking whether Γ is *k-satisfiable*, that is, if it admits a model which can be presented within k steps; checking whether a certain formula φ is *k-possible* with respect to Γ, corresponding to $\Gamma \wedge \varphi$ being k-satisfiable; and checking whether a certain formula φ is a necessary consequence (an assertion) for Γ, corresponding to $\Gamma \wedge \neg\varphi$ not being k-satisfiable. These problems can be easily reduced to checking the (bounded) satisfiability of a generic set of formulae (or, equivalently, of their conjunction). They can also be seen as a bounded model checking problem where the model is completely unspecified; compared to model checking, however, we notice that a model might not even be available at an early stage of the development process. This shift in focus makes the problems significantly different from a pragmatic point of view.

In this paper, we propose a new encoding of PLTL into propositional logic, based on the use of Separated Normal Form (SNF) for PLTL [8]. The main idea underlying the SNF reduction is the introduction of additional variables (subsequently referred to as 'SNF variables') to take into account the truth value of sub-formulae. The evolution of SNF variables is constrained by rules that can be seen as defining a transition relation of an observer automaton. The encoding can be enhanced further by considering that, in the bounded case, eventualities can be expressed with a fix-point construction. Our approach generalizes the construction of Frisch et al. [9], that shows significant improvements over the original construction presented in [4]. We carried out an experimental evaluation, where the SNF-based approach proposed in this paper is compared with the direct extension of BMC to past from [3]. The results show that the SNF approach results in a much more efficient implementation, yielding encodings that are smaller (in terms of clauses) and that are solved much more easily by the propositional solver.

This paper is structured as follows. Section 2 covers the syntax and semantics of PLTL. In Section 3 we introduce the Separated Normal Form for PLTL. In Section 4 we discuss how to generate efficient encodings for bounded model

checking of PLTL. Section 5 provides an experimental evaluation of our technique, and we draw some conclusions in Section 6.

2 Linear Temporal Logic with Past Operators

In this paper we consider PLTL, i.e. the Linear Temporal Logic (LTL) augmented with past operators. The starting point is standard LTL, the formulae of which are constructed from propositional atoms by applying the future temporal operators **X** (next), **F** (future), **G** (globally), **U** (until), and **R** (releases), in addition to the usual Boolean connectives. PLTL extends LTL by introducing the past operators **Y**, **Z**, **O**, **H**, **S**, and **T**, which are the temporal duals of the future operators and allow us to express statements on the past time instants. The **Y** (for "**Y***esterday*") operator is the temporal dual of **X** and refers to the *previous* time instant. At any non-initial time, $\mathbf{Y}\varphi$ is true if and only if φ holds at the previous time instant. The **Z** operator is similar to the **Y** operator, and it only differs in the way the initial time instant is dealt with: at time zero, $\mathbf{Y}\varphi$ is false, while $\mathbf{Z}\varphi$ is true.

The **O** (for "**O***nce*") operator is the temporal dual of **F** (sometimes in the future), so $\mathbf{O}\varphi$ is true iff φ is true at some past time instant (including the present time). Likewise, **H** (for "**H***istorically*") is the past-time version of **G** (always in the future), so that $\mathbf{H}\varphi$ is true iff φ is always true in the past. The **S** (for "**S***ince*") operator is the temporal dual of **U** (until), so that $\varphi\mathbf{S}\psi$ is true iff ψ holds somewhere in the past and φ is true from then up to now. Finally, we have $\varphi\mathbf{T}\psi = \neg(\neg\varphi\mathbf{S}\neg\psi)$ (**T** is called the "**T***rigger*" operator), exactly as in the future case we have $\varphi\mathbf{R}\psi = \neg(\neg\varphi\mathbf{U}\neg\psi)$.

The syntax of PLTL is formally defined as follows:

Definition 1 (Syntax of PLTL). *The grammar for PLTL formulae is*

$$PLTL \ni \varphi, \psi \doteq p \mid \neg\varphi \mid \varphi \circ^{\mathbf{B}} \psi \mid \circ_{\mathbf{1}}^{\mathbf{F}} \varphi \mid \varphi \circ_{\mathbf{2}}^{\mathbf{F}} \psi \mid \circ_{\mathbf{1}}^{\mathbf{P}} \varphi \mid \varphi \circ_{\mathbf{2}}^{\mathbf{P}} \psi$$

where $p \in \mathcal{A}$ and $\mathcal{A}$ is a finite set of atomic propositions, $\circ^{\mathbf{B}} \in \{\wedge, \vee\}$ stands for a Boolean connective, $\circ_{\mathbf{1}}^{\mathbf{F}} \in \{\mathbf{X}, \mathbf{F}, \mathbf{G}\}$ and $\circ_{\mathbf{2}}^{\mathbf{F}} \in \{\mathbf{R}, \mathbf{U}\}$ are future temporal operators (unary and binary, respectively), and $\circ_{\mathbf{1}}^{\mathbf{P}} \in \{\mathbf{Y}, \mathbf{Z}, \mathbf{O}, \mathbf{H}\}$ and $\circ_{\mathbf{2}}^{\mathbf{P}} \in \{\mathbf{T}, \mathbf{S}\}$ are past temporal operators (unary and binary).

In the following, we use φ and ψ to denote PLTL formulae, and p to denote propositions in $\mathcal{A}$. We write $\varphi \rightarrow \psi$ for $\neg\varphi \vee \psi$, and $\varphi \leftrightarrow \psi$ for $(\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$. As usual, PLTL formulae are interpreted over (linear) structures, that are basically infinite sequences of assignments to the propositions.

Definition 2 (Semantics of PLTL). *A* linear structure π *over a finite set of propositions $\mathcal{A}$ is a function $\pi : \mathbb{N} \rightarrow 2^{\mathcal{A}}$.*

Let π be a linear structure over $\mathcal{A}$, let φ and ψ be PLTL formulae, and let $i, j, k \in \mathbb{N}$. Then φ holds in π at time i, written $(\pi, i) \models \varphi$, is inductively defined in Figure 1. φ is true in π, written $\pi \models \varphi$, iff $(\pi, 0) \models \varphi$.

$$
\begin{array}{lll}
(\pi, i) \models p & \text{iff} & p \in \pi(i) \\
(\pi, i) \models \neg\varphi & \text{iff} & (\pi, i) \not\models \varphi \\
(\pi, i) \models \varphi \vee \psi & \text{iff} & (\pi, i) \models \varphi \text{ or } (\pi, i) \models \psi \\
(\pi, i) \models \varphi \wedge \psi & \text{iff} & (\pi, i) \models \varphi \text{ and } (\pi, i) \models \psi \\
 & & \\
(\pi, i) \models \mathbf{X}\,\varphi & \text{iff} & (\pi, i+1) \models \varphi \\
(\pi, i) \models \mathbf{F}\,\varphi & \text{iff} & \exists j \geq i.\ (\pi, j) \models \varphi \\
(\pi, i) \models \mathbf{G}\,\varphi & \text{iff} & \forall j \geq i.\ (\pi, j) \models \varphi \\
(\pi, i) \models \varphi\,\mathbf{U}\,\psi & \text{iff} & \exists j \geq i.\ ((\pi, j) \models \psi \text{ and } \forall k : i \leq k < j.\ (\pi, k) \models \varphi) \\
(\pi, i) \models \varphi\,\mathbf{R}\,\psi & \text{iff} & \forall j \geq i.\ ((\pi, j) \models \psi \text{ or } \exists k : i \leq k < j.\ (\pi, k) \models \varphi) \\
 & & \\
(\pi, i) \models \mathbf{Y}\,\varphi & \text{iff} & i > 0 \text{ and } (\pi, i-1) \models \varphi \\
(\pi, i) \models \mathbf{Z}\,\varphi & \text{iff} & i = 0 \text{ or } (\pi, i-1) \models \varphi \\
(\pi, i) \models \mathbf{O}\,\varphi & \text{iff} & \exists j \leq i.\ (\pi, j) \models \varphi \\
(\pi, i) \models \mathbf{H}\,\varphi & \text{iff} & \forall j \leq i.\ (\pi, j) \models \varphi \\
(\pi, i) \models \varphi\,\mathbf{S}\,\psi & \text{iff} & \exists j \leq i.\ ((\pi, j) \models \psi \text{ and } \forall k : j < k \leq i.\ (\pi, k) \models \varphi) \\
(\pi, i) \models \varphi\,\mathbf{T}\,\psi & \text{iff} & \forall j \leq i.\ ((\pi, j) \models \psi \text{ or } \exists k : j < k \leq i.\ (\pi, k) \models \varphi)
\end{array}
$$

Fig. 1. The semantics of PLTL

Although the use of past operators in LTL does not introduce expressive power, it may allow to express temporal properties in an exponentially more succinct manner [14]. On an informal (but very important) level, past operators allow us to formalize properties more naturally. For instance, *if a problem is diagnosed, then a failure must have previously occurred*, can be represented in PLTL as

$$\mathbf{G}(\mathit{problem} \rightarrow \mathbf{O}\ \mathit{failure})$$

that is more natural than its pure-future counterpart $\neg(\neg \mathit{failure}\,\mathbf{U}\,\mathit{problem})$. Similarly, the property *grants are issued only upon requests* can be easily specified as

$$\mathbf{G}(\mathit{grant} \rightarrow \mathbf{Y}(\neg \mathit{grant}\ \mathbf{S}\ \mathit{request}))$$

compared to the corresponding pure-future translation

$$(\mathit{request}\,\mathbf{R}\,\neg \mathit{grant})\ \wedge\ \mathbf{G}(\mathit{grant} \rightarrow (\mathit{request} \vee (\mathbf{X}(\mathit{request}\,\mathbf{R}\,\neg \mathit{grant})))).$$

As for the pure future case, any formula in PLTL can be reduced to *Negation Normal Form* (NNF), where negation only occurs in front of atomic propositions. This linear time transformation is obtained by pushing the negation towards the leaves of the syntactic tree of the formula, and exploiting the dualities between

conjunction and disjunction, **F** and **G**, **U** and **R**, **O** and **H**, and **S** and **T**. Notice that, in the case of previous time we have to rely on the two properties $\neg\mathbf{Y}\varphi \equiv \mathbf{Z}\neg\varphi$ and $\neg\mathbf{Z}\varphi \equiv \mathbf{Y}\neg\varphi$, which extend the single future-case rule $\neg\mathbf{X}\varphi \equiv \mathbf{X}\neg\varphi$ (we have both $\neg\mathbf{Y}\varphi \not\equiv \mathbf{Y}\neg\varphi$ and $\neg\mathbf{Z}\varphi \not\equiv \mathbf{Z}\neg\varphi$, because of their semantics at the initial time point). We write the transformation to NNF of a formula φ as $\textsc{Nnf}(\varphi)$.

3 Separated Normal Form for PLTL

The Separated Normal Form (SNF) [7] is a clause-like normal form for temporal logic, based on the Separation Theorem [11]. A formula in SNF has the general form

$$\mathbf{G}\left(\bigwedge_i (P_i \rightarrow F_i)\right)$$

where each implication $P_i \rightarrow F_i$, also referred to as a *rule*, relates some past time formula P_i to some future time formula F_i. Each rule has one of the following forms:

$$\mathbf{start} \rightarrow \bigvee_j l_j \qquad \bigwedge_i l_i \rightarrow \bigvee_j l_j \qquad \bigwedge_i l_i \rightarrow \mathbf{X}\bigvee_j l_j \qquad \bigwedge_i l_i \rightarrow \mathbf{F}\bigvee_j l_j$$

where l_i, l_j are literals (i.e. either atomic propositions or negations of atomic propositions), and **start** is an abbreviation for $\mathbf{Z}\bot$. In the following, the rules are referred to as start, invariant, next, and eventuality rules, respectively.

Every PLTL formula can be mapped onto a formula in SNF which is equisatisfiable [8]. With respect to [8], we generalize the form of the rules to permit general propositional formulae in place of $\bigwedge_i l_i$ and $\bigvee_j l_j$. A further slight difference is that we adopt a non-strict semantics for time operators, so that all temporal operators other than **X**, **Y** and **Z** take into account the present time instant. In order to reduce to SNF a generic PLTL formula γ, we define a transformation that manipulates sets of formulae. We start from the singleton set $\{\mathbf{start} \rightarrow \textsc{Nnf}(\gamma)\}$, which intuitively states that γ has to hold in the initial state of any satisfying structure. Then, the conversion is carried out by the function $\textsc{Snf}(\cdot)$, which takes in input a set of formulae, and applies some transformation to a member of the set. The function is applied repeatedly until a set of rules is obtained. Intuitively, the transformations are devoted to eliminating occurrences of "complex" temporal operators by reducing them to more basic ones (i.e. **X** and **F**). To this end, each transformation can introduce new SNF variables, one for each temporal sub-formula being eliminated. In order to highlight their intuitive meaning, SNF variables are denoted as underlined temporal formulae (e.g. $\underline{\mathbf{XG}\varphi}$).

The transformations defining $\textsc{Snf}(\cdot)$ are reported in Figures 2 and 3. We write Γ for the subset of formulae which are not affected by the transformation, φ and ψ for PLTL formulae in NNF, and f and g for propositional formulae. In the rule being transformed, φ is the sub-formula that is not affected. We also write

$$\textsc{Snf}_{[\mathbf{X}]}(\{\varphi \to \psi(\mathbf{X}\, f)\} \cup \Gamma) \doteq \left\{ \begin{array}{c} \varphi \to \psi(\underline{\mathbf{X}\, f}) \\ \underline{\mathbf{X}\, f} \to \mathbf{X}\, f \end{array} \right\} \cup \Gamma$$

$$\textsc{Snf}_{[\mathbf{F}]}(\{\varphi \to \psi(\mathbf{F}\, f)\} \cup \Gamma) \doteq \left\{ \begin{array}{c} \varphi \to \psi(\underline{\mathbf{F}\, f}) \\ \underline{\mathbf{F}\, f} \to \mathbf{F}\, f \end{array} \right\} \cup \Gamma$$

$$\textsc{Snf}_{[\mathbf{Y}]}(\{\psi(\mathbf{Y}\, f) \to \varphi\} \cup \Gamma) \doteq \left\{ \begin{array}{c} \psi(\underline{\mathbf{Y}\, f}) \to \varphi \\ \mathbf{Y}\, f \to \underline{\mathbf{Y}\, f} \end{array} \right\} \cup \Gamma$$

$$\textsc{Snf}_{[\mathbf{Z}]}(\{\psi(\mathbf{Z}\, f) \to \varphi\} \cup \Gamma) \doteq \left\{ \begin{array}{c} \psi(\underline{\mathbf{Z}\, f}) \to \varphi \\ \mathbf{Z}\, f \to \underline{\mathbf{Z}\, f} \end{array} \right\} \cup \Gamma$$

$$\textsc{Snf}_{[\mathbf{G}]}(\{\varphi \to \psi(\mathbf{G}\, f)\} \cup \Gamma) \doteq \left\{ \begin{array}{c} \varphi \to \psi(f \wedge \underline{\mathbf{X}(\mathbf{G}\, f)}) \\ \underline{\mathbf{X}(\mathbf{G}\, f)} \to \mathbf{X}\,(f \wedge \underline{\mathbf{X}(\mathbf{G}\, f)}) \end{array} \right\} \cup \Gamma$$

$$\textsc{Snf}_{[\mathbf{U}]}(\{\varphi \to \psi(f\, \mathbf{U}\, g)\} \cup \Gamma) \doteq \left\{ \begin{array}{c} \varphi \to \psi(g \vee (f \wedge \underline{\mathbf{X}(f\, \mathbf{U}\, g)})) \\ \underline{\mathbf{X}(f\, \mathbf{U}\, g)} \to \mathbf{X}\,(g \vee (f \wedge \underline{\mathbf{X}(f\, \mathbf{U}\, g)})) \\ \varphi \to \mathbf{F}\, g \end{array} \right\} \cup \Gamma$$

$$\textsc{Snf}_{[\mathbf{R}]}(\{\varphi \to \psi(f\, \mathbf{R}\, g)\} \cup \Gamma) \doteq \left\{ \begin{array}{c} \varphi \to \psi(g \wedge (f \vee \underline{\mathbf{X}(f\, \mathbf{R}\, g)})) \\ \underline{\mathbf{X}(f\, \mathbf{R}\, g)} \to \mathbf{X}\,(g \wedge (f \vee \underline{\mathbf{X}(f\, \mathbf{R}\, g)})) \end{array} \right\} \cup \Gamma$$

$$\textsc{Snf}_{[\mathbf{O}]}(\{\psi(\mathbf{O}\, f) \to \varphi\} \cup \Gamma) \doteq \left\{ \begin{array}{c} \psi(f \vee \underline{\mathbf{Y}(\mathbf{O}\, f)}) \to \varphi \\ \mathbf{Y}\,(f \vee \underline{\mathbf{Y}(\mathbf{O}\, f)}) \to \underline{\mathbf{Y}(\mathbf{O}\, f)} \end{array} \right\} \cup \Gamma$$

$$\textsc{Snf}_{[\mathbf{H}]}(\{\psi(\mathbf{H}\, f) \to \varphi\} \cup \Gamma) \doteq \left\{ \begin{array}{c} \psi(f \wedge \underline{\mathbf{Z}(\mathbf{H}\, f)}) \to \varphi \\ \mathbf{Z}\,(f \wedge \underline{\mathbf{Z}(\mathbf{H}\, f)}) \to \underline{\mathbf{Z}(\mathbf{H}\, f)} \end{array} \right\} \cup \Gamma$$

$$\textsc{Snf}_{[\mathbf{S}]}(\{\psi(f\, \mathbf{S}\, g) \to \varphi\} \cup \Gamma) \doteq \left\{ \begin{array}{c} \psi(g \vee (f \wedge \underline{\mathbf{Z}(f\, \mathbf{S}\, g)})) \to \varphi \\ \mathbf{Z}\,(g \vee (f \wedge \underline{\mathbf{Z}(f\, \mathbf{S}\, g)})) \to \underline{\mathbf{Z}(f\, \mathbf{S}\, g)} \end{array} \right\} \cup \Gamma$$

$$\textsc{Snf}_{[\mathbf{T}]}(\{\psi(f\, \mathbf{T}\, g) \to \varphi\} \cup \Gamma) \doteq \left\{ \begin{array}{c} \psi(g \wedge (f \vee \underline{\mathbf{Z}(f\, \mathbf{T}\, g)})) \to \varphi \\ \mathbf{Z}\,(g \wedge (f \vee \underline{\mathbf{Z}(f\, \mathbf{T}\, g)})) \to \underline{\mathbf{Z}(f\, \mathbf{T}\, g)} \end{array} \right\} \cup \Gamma$$

$$\textsc{Snf}_{[\mathbf{Y2X}]}(\{\mathbf{Y}\, f \to \varphi\} \cup \Gamma) \doteq \{f \to \mathbf{X}\, \varphi\} \cup \Gamma$$

$$\textsc{Snf}_{[\mathbf{Z2X}]}(\{\mathbf{Z}\, f \to \varphi\} \cup \Gamma) \doteq \left\{ \begin{array}{c} \mathbf{start} \to \varphi \\ f \to \mathbf{X}\, \varphi \end{array} \right\} \cup \Gamma$$

Fig. 2. Part of the transformation function for SNF.

$\psi(\mathbf{G}f)$ to say that $\mathbf{G}f$ occurs in ψ, while $\psi(g)$ stands for the formula obtained by substituting every occurrence of $\mathbf{G}f$ with g in ψ. The same notation is used for the other temporal operators. The first four transformations in Figure 2, $\textsc{Snf}_{[\mathbf{X}]}$, $\textsc{Snf}_{[\mathbf{F}]}$, $\textsc{Snf}_{[\mathbf{Y}]}$ and $\textsc{Snf}_{[\mathbf{Z}]}$ are used to rename sub-formulae. The others have an intuitive interpretation, based on the fix-point characterizations of temporal operators. Consider the simple case of a $\mathbf{G}f$ formula: the corresponding set of rules is $\{\mathbf{start} \to f \wedge \underline{\mathbf{XG}f}, \underline{\mathbf{XG}f} \to \mathbf{X}(f \wedge \underline{\mathbf{XG}f})\}$. The intuitive interpretation for the SNF variable $\underline{\mathbf{XG}f}$ is that $\mathbf{G}f$ holds in the next state. Similarly, consider

$$\textsc{Snf}_{[\text{p2p}]}(\{\mathbf{start} \rightarrow \varphi_P\} \cup \Gamma) \doteq \{\textsc{Nnf}(\neg\varphi_P) \rightarrow \neg\mathbf{start}\} \cup \Gamma$$
$$\textsc{Snf}_{[\text{f2f}]}(\{\psi_{\neg P} \rightarrow \neg\mathbf{start}\} \cup \Gamma) \doteq \{\mathbf{start} \rightarrow \textsc{Nnf}(\neg\psi_{\neg P})\} \cup \Gamma$$

$$\textsc{Snf}_{[\mathbf{startY}]}(\{\mathbf{start} \rightarrow \mathbf{Y}\,\varphi\} \cup \Gamma) \doteq \{\mathbf{start} \rightarrow \bot\} \cup \Gamma$$
$$\textsc{Snf}_{[\mathbf{startZ}]}(\{\mathbf{start} \rightarrow \mathbf{Z}\,\varphi\} \cup \Gamma) \doteq \{\mathbf{start} \rightarrow \top\} \cup \Gamma$$

Fig. 3. The transformation functions to deal with combining past and future

the rule $\mathbf{O}(f) \rightarrow g$: the corresponding set of rules is $\{f \vee \mathbf{YO}f \rightarrow g,\ f \vee \underline{\mathbf{YO}f} \rightarrow \mathbf{X}\,\underline{\mathbf{YO}f}\}$. The intuition here is that the SNF variable $\underline{\mathbf{YO}f}$ will hold in the next state if f holds in the current state, or it held in some previous state. It is easy to see that the above transformations only introduce SNF variables, and $\mathbf{F}, \mathbf{X}, \mathbf{Y}$ and $\mathbf{Z}$ operators; together, $\textsc{Snf}_{[\mathbf{Y2X}]}$ and $\textsc{Snf}_{[\mathbf{Z2X}]}$ replace previous operators with next operators, so that the only remaining operators are $\mathbf{F}$ and $\mathbf{X}$.

The transformations in Figure 2 rely on past operators appearing on the left side of rules and future operators on the right. The transformations $\textsc{Snf}_{[\text{p2p}]}$ and $\textsc{Snf}_{[\text{f2f}]}$, reported Figure 3, are used to move operators onto the appropriate side (we use φ_P to denote a PLTL formula with at least an occurrence of a past temporal operator applied to a purely propositional formula, and $\varphi_{\neg P}$ to denote a formula with no such occurrences). The other transformations in Figure 3 avoid renaming $\mathbf{Y}$ and $\mathbf{Z}$ operators in trivial cases.

In order to guarantee the termination of the transformation described above, some syntactic restrictions need to be enforced. The application of $\textsc{Snf}_{[\mathbf{F}]}$ is forbidden in cases where the $\mathbf{F}$ operator is the main connective of the conclusion, i.e. when the transformed rule has the form $\psi \rightarrow \mathbf{F}g$; similar restrictions apply to $\textsc{Snf}_{[\mathbf{X}]}$, $\textsc{Snf}_{[\mathbf{Y}]}$, and $\textsc{Snf}_{[\mathbf{Z}]}$. Furthermore, transformations $\textsc{Snf}_{[\mathbf{Y2X}]}$ and $\textsc{Snf}_{[\mathbf{Z2X}]}$ must not be used while the right hand side is $\neg\mathbf{start}$.

4 Encoding Bounded Verification of PLTL into SAT

Traditionally, temporal logics are used to express requirements over designs, represented as Kripke structures.

Definition 3. *A* (Boolean) Kripke structure *over $\mathcal{A}$ is a tuple $M = \langle S, I, T\rangle$, where $S = 2^{\mathcal{A}}$ is a finite set of states, $I \subseteq S$ is the set of initial states, $T \subseteq S \times S$ is a transition relation between states. A path in M is an infinite sequence of states $s_0, s_1, \ldots$ such that $s_0 \in I$ and, for all i, $T(s_i, s_{i+1})$. Given a path $s_0, s_1, \ldots$, the corresponding linear structure maps i to s_i, for every i. A formula φ is* existentially *valid in M ($M \models \mathbf{E}\varphi$) iff it is true in the linear structure associated to some path π in M. Conversely, φ is* universally *valid in M ($M \models \mathbf{A}\varphi$) iff it is true in every linear structure associated to a path in M.*

Clearly, there is a duality between the existential and the universal versions of the model checking problem, i.e. $M \models \mathbf{A}\varphi$ iff $M \not\models \mathbf{E}\neg\varphi$. The universal model checking problem can be intuitively interpreted as checking if all the behaviors in the system represented by M comply with the requirement φ; the existential version is often interpreted as the problem of finding a witness to a violation of a required property. In the following, we assume that a Kripke structure M is given, and do not distinguish between a path in M and the corresponding linear structure. The satisfiability problem for φ can be seen as a model checking problem $M \models \varphi$, where M is a completely unconstrained Kripke structure of the form $\langle S, S, S \times S\rangle$, with $S = 2^{\mathcal{A}}$, and $\mathcal{A}$ is the set of atomic propositions in φ.

Bounded Verification The idea underlying bounded verification is to look for linear structures that can be presented with a number of steps (i.e. transitions) which is fixed a priori. We assume that the number of steps, also called the bound, is denoted k and given. While completeness may be lost, the exploitation of the bound often enables the use of alternate search techniques. The idea of Bounded Model Checking [4] is to reduce an existential model checking problem $M \models \varphi$ with bound k to the problem of checking the satisfiability of a propositional formula $[\![M \models_k \varphi]\!]$: this is satisfiable iff there exists a path in M which can be presented with k transitions and satisfies φ. The encoding is structured as a conjunction $\textsc{Path}_k \wedge [\![\varphi]\!]_k$, where the (propositional) models of the first conjunct correspond to finitely-expressible paths in M, while the second component encodes the requirements induced by φ. In the following, we assume that $\mathcal{A}$ is the set of atomic propositions occurring in M and in φ. We do not address the construction for $\textsc{Path}_k$, which is standard. The case of bounded satisfiability simply reduces to the case of bounded model checking by simply dropping the $\textsc{Path}_k$ component from the encoding.

The problem of bounded satisfiability for φ is reduced to a propositional satisfiability problem as follows. The language of the propositional theory is defined by introducing, for each atomic proposition p in $\mathcal{A}$, $k+1$ propositional variables of the form $p(i)$, with i ranging from 0 to k. When the propositional variable $p(i)$ is assigned to true [false, respectively], the intuitive meaning is that p holds [does not hold] in the i-th state of the linear structure. In addition, the language of the propositional theory contains, for each SNF variable associated to $\textsc{Snf}(\varphi)$, $k+1$ propositional variables.

Intuitively, with bounded verification, it is possible to encode two different kinds of linear structures for φ: without loops, and with loops. When no loop is required, the propositional model corresponds to a whole class of linear structures sharing the same finite prefix, and which is sufficient to show the satisfiability of the formula φ. Intuitively, this is the case of violations to safety properties, which require that nothing bad ever happens – and it is therefore sufficient to show a finite path leading to a bad situation. When a loop is required, the propositional model corresponds to a lasso-shaped linear structure, which is made up of a finite prefix u followed by a portion v repeated infinitely many times. Intuitively, this

is the case of violations to liveness properties, which requires that something good should happen. In this case, the structure reaches a point where only bad states keep repeating. While the case of a "finite" prefix requires no additional constraints, in order to find a looping behavior we enforce that the k-th state be equal to same preceding state. In the propositional theory, a loop-back from k to l, with $l < k$, is captured by stating that, for each atomic proposition $p \in \mathcal{A}$, the corresponding propositional variables at k and l are assigned the same truth values, i.e. ${}_lL_k \doteq \bigwedge_{p\in\mathcal{A}}(p(l) \leftrightarrow p(k))$.

Encoding the SNF Rules The problem of k-satisfiability for a PLTL formula φ is obtained by encoding each rule in $\textsc{Snf}(\varphi)$ over the $k+1$ time instants, depending on the existence of a loop. The encoding is structured as follows:

$$\bigwedge_{i=0}^{k} \bigwedge_{\rho\in\textsc{Snf}(\varphi)} {}_-[\![\rho]\!]_k^i \quad \vee \quad \bigvee_{l=0}^{k-1} \left({}_lL_k \wedge \bigwedge_{i=0}^{k} \bigwedge_{\rho\in\textsc{Snf}(\varphi)} {}_l[\![\rho]\!]_k^i \right)$$

where ${}_l[\![\cdot]\!]_i^k$ stands for the encoding operator over a path of k steps, at step i, with loop-back at l. We use $l \in \mathbb{N}$ to denote the loop-back point, while $l = -$ denotes the absence of a loop. The rules are encoded as follows:

$$
\begin{aligned}
{}_l[\![\mathbf{start} \rightarrow f]\!]_k^i &\doteq \begin{cases} {}_l[\![f]\!]_k^i & \text{if } i = 0 \\ \top & \text{otherwise} \end{cases} \\
{}_l[\![f \rightarrow g]\!]_k^i &\doteq {}_l[\![f]\!]_k^i \rightarrow {}_l[\![g]\!]_k^i \\
{}_l[\![f \rightarrow \mathbf{X}\, g]\!]_k^i &\doteq \begin{cases} {}_l[\![f]\!]_k^i \rightarrow {}_l[\![g]\!]_k^{i+1} & \text{if } i < k \\ {}_l[\![f]\!]_k^i \rightarrow {}_l[\![g]\!]_k^{l+1} & \text{if } i = k \text{ and } l \in \mathbb{N} \\ {}_l[\![f]\!]_k^i \rightarrow \bot & \text{if } i = k \text{ and } l = - \end{cases} \\
{}_l[\![f \rightarrow \mathbf{F}\, g]\!]_k^i &\doteq \begin{cases} {}_-[\![f]\!]_k^i \rightarrow {}_-[\![g \vee \underline{\mathbf{XF}g}]\!]_k^i \quad \wedge & \\ \quad {}_-[\![\underline{\mathbf{XF}g} \rightarrow \mathbf{X}(g \vee \underline{\mathbf{XF}g})]\!]_k^i & \text{if } l = - \\ {}_-[\![f]\!]_k^i \rightarrow {}_-[\![g \vee \underline{\mathbf{XF}g}]\!]_k^{min(i,l)} \quad \wedge & \\ \quad {}_l[\![\underline{\mathbf{XF}g} \rightarrow \mathbf{X}(g \vee \underline{\mathbf{XF}g})]\!]_k^i & \text{if } l \in \mathbb{N} \end{cases}
\end{aligned}
$$

Intuitively, the rules are expanded as follows. The start rules express constraints only on the initial situation, and therefore have no effect on the subsequent time points. The invariant rules equally affect all of the time instants. The next rules are encoded in three different ways, depending on k, i, and l. Before the last state, the expansion is independent of l and k: the premise f is codified at state i, and the matrix of the conclusion g at $i+1$. At the last state, the premise is codified at k, while the matrix of the conclusion is either expanded at $l+1$, when a loop exists, or reduces to false, in case of no loop-back. The expansion of the eventuality rule requires the preliminary creation of an SNF variable, $\underline{\mathbf{XF}g}$, representing the fact that the eventuality is to be fulfilled at

next state. Then, in the case of no loop-back, the expansion basically performs a renaming, generating an invariant rule, and a next rule describing the dynamics together with the enforcement of the eventuality before the end of the path. This description expresses the loop optimization obtained in [9] with the introduction of the bound operator. The loop case is reduced to the case without a loop at $min(i, l)$: this encompasses both the possibility of $i \geq l$, i.e. i is in the loop, and of $i < l$, i.e. l is before the loop.

The expansion of purely propositional formulae is straightforward. Notice however that their conversion may impact the way in which the corresponding CNF is obtained, and therefore on the efficiency of the SAT solver. For lack of space we do not address these issues here (see e.g. [16]).

The number of propositional variables in the encoding is $O((|\mathcal{A}| + n) \cdot k)$, where n is the number of occurrences of temporal operators in φ. In fact, each transformation introduces one new SNF variable, and each temporal operator can result in the introduction of up to two new variables. The worst case is the **U** operator, that requires the application of $\text{SNF}_{[\mathbf{U}]}$, with the encoding for **F** introducing a second variable. We also notice that the number of rules in $\text{SNF}(\varphi)$ is linear in n: for each occurrence of a temporal operator, SNF applies exactly one transformation, which can in turn require the application of another transformation. The worst case is again associated with the expansion of **U**. The number of rule instances in the above encoding is $O(n \cdot k^2)$, because of the different loop-back points.

Loop Independence Optimization In order to overcome the quadratic dependence on k, we further develop the encoding, arriving at a formulation with a number of rule instances that is $O(n \cdot k)$. We exploit the fact that the encoding for most of the rules can be written to be the same in both the loop and non-loop cases, and we explicitly factor it out. This is obtained by rewriting the rules in a way that is independent of the actual existence and position of a loop-back, and by factoring them out of the big disjunction over the possible loop-back points. The encoding is structured as follows:

$$\bigwedge_{i=0}^{k-1} \bigwedge_{\rho \in \text{SNF}(\varphi)} {}^{\text{LI}}[\![\rho]\!]_k^i \;\wedge\; \left(\bigwedge_{\rho \in \text{SNF}(\varphi)} {}_{-}^{\text{LD}}[\![\rho]\!]_k^k \;\vee\; \bigvee_{l=0}^{k-1} \left({}_l L_k \wedge \bigwedge_{\rho \in \text{SNF}(\varphi)} {}_{l}^{\text{LD}}[\![\rho]\!]_k^k \right) \right)$$

where ${}^{\text{LI}}[\![\cdot]\!]_k^i$ and ${}^{\text{LD}}[\![\cdot]\!]_k^i$ denote the *loop-independent* encoding and the *loop-dependent* encoding operators. The definition of ${}^{\text{LI}}[\![\cdot]\!]_k^i$ for the start, invariant and next rules coincides with ${}_{-}[\![\cdot]\!]_k^i$. For the eventuality rule $f \rightarrow \mathbf{F}g$, we first notice that the dependence on l in $min(i, l)$, in the loop case, can be eliminated with a disjunction of the encodings at i and at l. That is, ${}_l[\![\mathbf{F}\, g]\!]_k^i$ is replaced by ${}_l[\![\mathbf{F}\, g]\!]_k^i \vee {}_l[\![\mathbf{F}\, g]\!]_k^l$. The factorization is completed by renaming every occurrence of ${}_l[\![\mathbf{F}\, g]\!]_k^l$ with a newly introduced variable $\text{ATL}(\underline{\mathbf{F}g})$. The same variable is disjuncted to ${}_{-}[\![\mathbf{F}\, g]\!]_k^i$ in the case without a loop. The encoding thus becomes, regardless of

the loop-back point,

$$ {}^{\text{LI}}[\![f \rightarrow \mathbf{F}\, g]\!]^i_k \doteq \begin{cases} {}_-[\![f]\!]^i_k \rightarrow ({}_-[\![g \vee \underline{\mathbf{XF}g}]\!]^i_k \vee \text{AtL}(\underline{\mathbf{F}g})) & \wedge \\ \quad {}_-[\![\underline{\mathbf{XF}g} \rightarrow \mathbf{X}(g \vee \underline{\mathbf{XF}g})]\!]^i_k \end{cases} $$

The encoding of the loop-dependent part for the start, invariant and next rules coincides with the encoding operator defined in previous section. (For the sake of clarity, we do not make explicit the fact that the invariant rules are independent of the loop, and could therefore be factored out; this fact is however exploited in the implementation.) The case of eventuality is encoded as follows.

$$ {}^{\text{LD}}_{l}[\![f \rightarrow \mathbf{F}\, g]\!]^k_k \doteq \begin{cases} {}_-[\![f]\!]^k_k \rightarrow \neg\text{AtL}(\underline{\mathbf{F}g}) & \wedge \\ \quad {}_-[\![\underline{\mathbf{XF}g} \rightarrow \mathbf{X}(g \vee \underline{\mathbf{XF}g})]\!]^k_k & \text{if } l = - \\ ({}_-[\![f]\!]^k_k \wedge \text{AtL}(\underline{\mathbf{F}g})) \rightarrow {}_-[\![g \vee \underline{\mathbf{XF}g}]\!]^l_k & \wedge \\ \quad {}_l[\![\underline{\mathbf{XF}g} \rightarrow \mathbf{X}(g \vee \underline{\mathbf{XF}g})]\!]^k_k & \text{if } l \in \mathbb{N} \end{cases} $$

We remark that "AtL" variables are untimed: unlike the variables in φ and from the Snf variables, they are not replicated $k+1$ times. We achieve independence from the loop since different characterising clauses are activated, depending on the particular value of l.

5 Experimental Analysis

In this section, we compare the SNF approach with the method for bounded model checking for PLTL proposed in [3], hereafter referred to as the *direct encoding*, that is a generalisation of the encoding for LTL [4]. The direct encoding is defined by recursively descending the structure of the formula being encoded, and distinguishing between the case without a loop and the case with a loop. In the case without a loop, the truth of a PLTL formula only depends on the finite prefix, and the interpretation of past operators always progresses towards the points closer to the origin (i.e., from i to 0). In the case of the loop, the problem is significantly more complicated: in fact, when interpreting a PLTL formula within the loop, the interpretation of going into the past may correspond either to going into the prefix before the loop-back point, or back to the future. The problem is solved by introducing the notion of past temporal horizon of a formula, that is then used as an upper bound to the number of virtual unrolls needed when generating the encoding for the formula. Similar to the pure-future case, the direct encoding does not introduce additional variables, so that witnesses of the form $\alpha \cdot \beta^k \cdot \beta^\omega$ can be reached with $k = |\alpha| \cdot |\beta|$ steps.

Both methods were implemented in NuSMV [6]. For each problem instance, and for each method, we report the total time required by NuSMV (on a Pentium 4, 1.8GHz processor with 1Gb RAM) to build and solve the encodings up to the reported bound, using zChaff [15] as the SAT solver; the reported bound corresponds to the first satisfiable instance, or to the largest unsatisfiable instance solved within the time limit.

	Counter(16)		Counter(32)		Counter(64)	
	Direct	SNF	Direct	SNF	Direct	SNF
$P(0)$	0.07 8	0.06 8	0.5 16	0.19 16	8.10 32	0.96 32
$P(1)$	8.43 17	0.27 17	680.94 33	1.50 33	T.O. 37	11.20 65
$P(2)$	256.99 17	1.03 26	T.O. 21	7.33 50		68.60 98
$P(3)$	T.O. 13	3.27 35		27.73 67		282.01 131
$P(4)$		8.89 44		81.59 84		966.92 164

Table 1. The results for Counter(N).

We first ran the test from [3] involving past operators, i.e. the Alternating Bit Protocol (from the NuSMV distribution) with a property of the form

$$\mathbf{G}(\texttt{sender.state} = \texttt{waitForAck} \rightarrow \mathbf{Y}\,\mathbf{H}\,\texttt{sender.state} \neq \texttt{waitForAck})$$

The direct encoding required 87.2 secs. to generate the encoding and solve the problem, while the SNF-based encoding requires only 56.2 secs. Both methods find a counterexample at depth 17.

In order to stress the ability of the two methods to process past operators and to find short counterexamples, we conceived the Counter(N) problem set: a counter starts at 0, progresses up to N, and then loops back at $N/2$. We evaluate a set of parameterized properties, of the form

$$P(i) \doteq \neg\mathbf{F}(\mathbf{O}((c = N/2) \wedge \mathbf{O}((c = N/2 + 1) \ldots \wedge \mathbf{O}(c = N/2 + i) \ldots)))$$

The value of i is a measure of the nesting of past operators, while the structure of the property requires that the loop (of length $N/2$) must be traversed backwards several times in order to reach a counterexample.

The results are reported in Table 1, where T.O. indicates a runtime exceeding 1800 secs. The direct encoding suffers from the nesting of the property, which influences the past temporal horizon and therefore requires a larger number of virtual unrolls. Most of the time is in fact spent in the generation of the encodings. On the contrary, the encodings are generated efficiently by the SNF-based method, and the time required by the SAT solver is also very limited. SNF-based encodings seem to yield a significant speed up, even if longer paths need to be explored in order to find a counterexample. Notice however that in this problem set the component related to the model is not very significant. Although the ability to construct counterexamples with virtual unroll of the past might be a win, there is clearly a tradeoff between the time that is saved in searching shortened counterexamples compared to the time that is invested in generating more complex encodings.

As a further step, we compared the SNF and the direct encodings on a test set from the domain of requirement engineering for software systems. The starting point is a description of a real-world scenario written in Formal Tropos [10], a language for the description of early requirements. The test set is obtained by conversion from the Formal Tropos model, parameterized in the number of

	Size 1		Size 1.5		Size 2	
PropType	Direct	SNF	Direct	SNF	Direct	SNF
EXISTS	0.10 1	0.46 1	1.00 1	2.69 1	32.57 1	45.92 1
POSS_1	0.09 2	0.52 2	1.59 2	3.12 2	42.62 2	53.02 2
POSS_2	0.08 2	0.52 2	1.55 2	3.19 2	43.20 2	52.88 2
POSS_3	0.13 3	0.62 3	2.94 3	3.85 3	64.67 3	63.00 3
POSS_4	0.10 2	0.54 2	1.47 2	3.15 2	42.72 2	53.29 2
POSS_5	0.12 3	0.60 3	2.95 3	3.95 3	66.11 3	63.87 3
POSS_6	18.61 20	7.77 20	1.50 2	3.19 2	41.80 2	52.69 2
POSS_7	18.78 20	8.10 20	0.91 20	2.66 20	32.39 20	45.88 20
POSS_8	19.36 20	7.86 20	1.92 2	3.28 2	43.23 2	53.80 2
POSS_9	0.11 2	0.52 2	1.58 2	3.14 2	41.92 2	53.97 2
POSS_10	21.55 20	10.69 20	2.96 3	3.83 3	64.11 3	63.76 3
POSS_11	0.16 3	0.60 3	2.98 3	3.83 3	66.01 3	63.03 3
POSS_12	22.21 20	8.34 20	T.O. 16	559.50 20	T.O. 9	T.O. 13
ASS_1	21.36 20	9.22 20	T.O. 16	851.10 20	T.O. 9	T.O. 12
ASS_2	21.44 20	9.04 20	T.O. 16	217.08 20	T.O. 9	T.O. 18
ASS_3	22.44 20	9.71 20	T.O. 16	192.77 20	42.31 2	52.98 2
ASS_4	21.72 20	10.70 20	1.54 2	3.12 2	44.38 2	56.29 2
ASS_5	20.59 20	8.80 20	T.O. 16	217.87 20	T.O. 9	T.O. 17
ASS_6	17.91 20	7.89 20	T.O. 16	173.54 20	T.O. 9	1730.54 20
ASS_7	17.52 20	7.81 20	T.O. 16	197.76 20	T.O. 9	T.O. 16
ASS_8	21.70 20	8.62 20	T.O. 16	504.25 20	T.O. 9	T.O. 13
ASS_9	21.12 20	10.69 20	T.O. 16	363.21 20	T.O. 9	T.O. 14
ASS_10	21.51 20	9.50 20	T.O 16	840.48 20	T.O. 9	T.O. 12
ASS_11	20.77 20	11.42 20	T.O. 16	114.16 20	T.O. 9	T.O. 15
ASS_12	21.81 20	10.75 20	T.O. 16	142.81 20	T.O. 9	1779.20 20

Table 2. The results on the examples from [10].

instances for each class in the model, to a set of (ground) PLTL formulae. The parameterization sets the number of instances with which each class in the description is populated. Different kinds of checks are performed, ranging from feasibility of built-in or domain-specific properties (EXISTS and POSS), for which witnesses are sought, and assertion violations (ASS), for which counterexamples are sought[3].

The results are reported in Table 2. We tackle problems for three degrees of instantiation: Size 1 corresponds to one object per class; Size 1.5 corresponds to the instantiation of one object for some classes and two objects for the remaining ones; in Size 2, each class is instantiated twice. The first column identifies the problem; three sets of columns follow, one for each size instantiation. T.O. indicates that the run-time exceeded 1800 secs. The maximum bound was set to 20. The instances which reached the maximal bound or timed out are unsatisfiable. The reported bound represents the length of the witness (for POSS

[3] More details on the Formal Tropos problem set can be found at `http://sra.itc.it/tools/t-tool/experiments/cm/`

and EXISTS), or of the counterexample (for ASS); or, in case of a timeout, the depth of the largest k for which the analysis was completed.

The results show that, on this class of problems, the direct encoding is somewhat superior on easier instances which are satisfiable with a small bound. However, on the harder instances, often requiring the exploration to higher bounds, the gain obtained by means of the SNF encoding with respect to the direct encoding is uniform. For the hardest problem instances, the speed up becomes very significant, sometimes bigger than an order of magnitude. The use of SNF also allows problems to be tackled that were previously out of reach within the time limit; when both methods time out, SNF is uniformly able to cover problem instances with higher length.

6 Conclusions and Future Work

In this paper, we have proposed the use of Separated Normal Form for the generation of encodings for bounded verification of Linear Temporal Logic with Past. We have shown the effectiveness of the approach by an experimental comparison with the previously available direct method [3], where our SNF-based approach is able to gain up to one order of magnitude.

The SNF transformation appears to bring the benefits of a pure future encoding without the usual exponential blowup associated with past to future transformations; this is believed to be a result of the bounded nature of the encoding, and future work will examine fully the theoretical implications of this. For the experimental work, the similarity between SNF and alternating automata calls for a comparison with this, and other, automata techniques.

Broadening the scope of the work, we expect that the techniques presented will be amenable to SAT-based induction in order to achieve completeness. Similarly, the SNF encoding is particularly suitable for use with incremental SAT solvers. These systems have proved useful for bounded model checking to reduce the amount of work involved in iterating up to a bound; for requirements verification the amount of repeated work currently necessary when testing multiple formulae with respect to a set of requirements will be reduced. Finally, we plan to extend the work to make use of non-Boolean SAT solvers to avoid the Booleanization of the data paths.

References

1. Accellera. *Accelera Property Specification Language: Reference Manual — Version 1.0.*
2. F. Bacchus and F. Kabanza. Control strategies in planning. In *Proceedings of the AAAI Spring Symposium Series on Extending Theories of Action: Formal Theory and Practical Applications*, pages 5–10, Stanford University, CA, USA, March 1995.
3. M. Benedetti and A. Cimatti. Bounded model checking for past LTL. In *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS'03*, Lecture Notes in Computer Science, Warsaw, Poland, April 2003. Springer-Verlag.

4. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In W.R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems. 5th International Conference, TACAS'99*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, July 1999.
5. J. Castro, M. Kolp, and J. Mylopoulos. A requirements-driven development methodology. In *Proceedings of the 13th International Conference on Advanced Information Systems Engineering*, 2001.
6. A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Proceedings of the Eleventh Conference on Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 495–499, Trento, Italy, July 1999. Springer-Verlag.
7. M. Fisher. A resolution method for temporal logic. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI)*. Morgan Kaufmann, August 1991.
8. M. Fisher and P. Noël. Transformation and synthesis in MetateM Part I: Propositional MetateM. Technical Report UMCS-92-2-1, Department of Computer Science, University of Manchester, Manchester M13 9PL, England, February 1992.
9. A. Frisch, D. Sheridan, and T. Walsh. A fixpoint based encoding for bounded model checking. In M D Aagaard and J W O'Leary, editors, *Formal Methods in Computer-Aided Design; 4th International Conference, FMCAD 2002*, volume 2517 of *Lecture Notes in Computer Science*, pages 238–254, Portland, OR, USA, November 2002. Springer-Verlag.
10. A. Fuxman, L. Liu, , M. Pistore, M. Roveri, and J. Mylopoulos. Specifying and analyzing early requirements in Tropos: Some experimental results. In *Proceedings of the* 11^{th} *IEEE International Requirements Engineering Conference*, Monterey Bay, California USA, September 2003. ACM-Press.
11. D. Gabbay. The declarative past and imperative future. In H. Barringer, editor, *Proccedings of the Colloquium on Temporal Logic and Specifications*, volume 398 of *Lecture Notes in Computer Science*, pages 409–448. Springer-Verlag, 1989.
12. S. Gnesi, D. Latella, and G. Lenzini. Formal verification of cryptographic protocols using history dependent automata. In *Proceedings of the of the 4th Workshop on Sistemi Distribuiti: Algoritmi, Architetture e Linguaggi*, 1999.
13. O. Kupferman, N. Piterman, and M. Vardi. Extended temporal logic revisited. In *Proceedings of the 12th International Conference on Concurrency Theory*, number 2154 in Lecture Notes in Computer Science, pages 519–534. Springer Verlag, 2001.
14. F. Laroussinie, N. Markey, and Ph. Schnoebelen. Temporal logic with forgettable past. In *Proceedings of the 17th IEEE Symp. Logic in Computer Science (LICS'2002)*, pages 383–392, Copenhagen, Denmark, July 2002. IEEE Comp. Soc. Press.
15. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *39th Design Automation Conference*, Las Vegas, June 2001.
16. D. Sheridan. The optimality of a fast CNF conversion and its use with SAT. Technical Report APES-82-2002, APES Research Group, March 2004. Available from `http://www.dcs.st-and.ac.uk/~apes/apesreports.html`.
17. A. van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proceedings of the 5th IEEE International Symposium on Requirements Engineering*, pages 249–263, 2001.

Clause Form Conversions for Boolean Circuits

Paul Jackson and Daniel Sheridan

School of Informatics
University of Edinburgh
Edinburgh, UK
pbj@inf.ed.ac.uk
d.j.sheridan@sms.ed.ac.uk

Abstract. Boolean circuits are well established as a data structure for building propositional encodings of problems in preparation for satisfiability solving. The standard method for converting Boolean circuits to clause form (naming every vertex) has a number of shortcomings
In this paper we give a projection of several well-known clause form conversions to a simplified Boolean circuit. We introduce a new conversion which we show is equivalent to that of Boy de la Tour in certain circumstances and is hence optimal in the number of clauses that it produces. We extend the algorithm to cover reduced Boolean circuits, a data structure used by the model checker NuSMV.
We present experimental results for this and other conversion procedures on BMC problems demonstrating its superiority, and conclude that the CNF conversion plays a significant role in reducing the overall solving time.

1 Introduction

SAT solvers based on the DPLL procedure typically require their input to be in conjunctive normal form (CNF). Earlier papers dealing with encoding to SAT, particularly much of the planning literature, encode directly from the input representation to clause form. More recent encoding work makes little mention of CNF conversion. Biere et al., proposing BMC [3], give an encoding to propositional logic only. Similarly, although the SNF encoding for BMC [6] discusses the clauses generated, the majority of the presentation in in general propositional logic. The microprocessor verification work of Velev includes a thorough analysis of improving the clause form generated [11], but the work is not immediately applicable to general propositional logic. Nevertheless, Velev is able to claim a speed up by a factor of 32 by altering the clause form conversion.

There is other evidence to motivate the study of clause form conversions for SAT. While focussing on CNF representations of cardinality constraints, Bailleux and Boufkhad [2] give a reformulation of the parity problems which have been standard SAT benchmarks for a number of years. They argue that the problems are made harder than they should be by a poor clause form representation, and demonstrate a dramatic speedup on the par32 problem with modern solvers on the reformulated problem.

In the first-order logic domain, the CNF conversion problem was handled comprehensively by Boy de la Tour [4]. The algorithm given is impractical without the improvements by Nonnengart et al. [9], but the resulting algorithm is fiddly to implement making it hard to be confident of a correct implementation.

In this paper we introduce a simple and easy to understand CNF conversion algorithm for propositional logic and prove that it is optimal with respect to the number of clauses. As its time complexity is linear, it represents a significant improvement over the (quadratic) Boy de la Tour algorithm. Of course, it is well known that problem size does not necessarily correspond to solving time in SAT, so we present some experimental results demonstrating the effect that our algorithm has on some BMC [3] problems.

1.1 Notation conventions

In an attempt to improve the clarity of the presentation, we use a number of conventions in our notation. Much of the work is concerned with both graphs and propositional logic, so we distinguish between *graph variables* ranging over vertices and edges given in italic capitals (X, Y) and *propositional variables* given in italic lower case (x, y); vertices are typically denoted V and edges E and this notation is significant in determining the type of a function. We will use the shorthand of referring to a subgraph by a single edge; the subgraph thus identified includes all of the descendents of the edge given, and such an edge is called the *root* of the subgraph and denoted T. Sets of vertices or edges are given in bold type ($\mathbf{X}$, $\mathbf{Y}$).

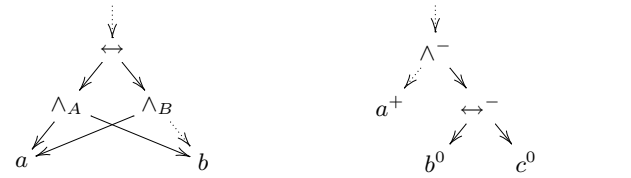
Where a function creates new propositional variables, these are given the name x_i where i is some identifier (typically a graph vertex). These variables are assumed to be unused in any other context.

2 Boolean Circuits

In contrast to the formulaic representation of propositional logic normally used, Boolean circuits are much closer to an electronics view of logic. Labelled input *wires* take the place of variables and together with (possibly unlabelled) internal wires they are connected by logic *gates* which compute various logic functions. This makes it very natural for the results of sub-circuits to be shared amongst other parts of the circuit, as would be expected in the physical world.

Boolean circuits may be efficiently represented as directed acyclic graphs (DAGs). Vertices having outgoing edges correspond to gates, with the edges pointing to the inputs to the gate. Vertices without outgoing edges (which we will call *leaf* vertices) are the input or output variables for the circuit.

Abdulla, Bjesse, and Eén proposed *reduced* Boolean circuits (RBCs) [1] as a DAG representation of a propositional formula with additional restrictions on the type and relationships of the gates which place RBCs somewhere between being a normal form and a canonical form for propositional formulæ. One of the key strengths of Boolean circuits is the ability to use one circuit to represent



(a) $a \wedge b \leftrightarrow (a \rightarrow b)$ reduced to $\neg(a \wedge b \leftrightarrow (a \wedge \neg b))$

(b) $a \vee (b \leftrightarrow \neg c)$ reduced to $\neg(\neg a \wedge (b \leftrightarrow c))$

Fig. 1. Example RBCs showing vertex labelling

a formula both positively and negatively. To preserve this property, Abdulla et al. eschew NNF in favour of restricting gates to conjunctions and equivalences (bi-implications), marking negation on the edges of the graph.

Definition 1 *An RBC is a DAG consisting of edges* $\mathbf{E}$ *and vertices* $\mathbf{V} = \mathbf{V_I} \cup \mathbf{V_L}$ *where internal vertices* $\mathbf{V_I}$ *represent operators, and leaf vertices* $\mathbf{V_L}$ *represent variables. The following properties are required to hold and form the encoding of Boolean circuits as DAGs:*

- *Each* $V \in \mathbf{V_I}$ *consists of an operator* $op(v) \in \{\wedge, \leftrightarrow\}$ *and a left and right edge* $(left(V), right(V) \in \mathbf{E})$.
- *Each* $V \in \mathbf{V_L}$ *contains a variable* $var(V)$.
- *Each* $E \in \mathbf{E}$ *has a sign* $sign(E) \in \{+, -\}$ *and a target vertex* $target(E) \in \mathbf{V}$.

The sign attribute encodes negation, where $sign(E) = +$ *indicates an unnegated edge and* $sign(E) = -$ *indicates a negated edge.*

The following additional properties serve to reduce the number of representations possible for equivalent formulæ:

- *All common subformulæ are shared:* $\forall V, V' \in \mathbf{V_I}, left(V) = left(V') \wedge right(V) = right(V') \rightarrow V = V'$.
- *The constant* $\top$ *only occurs in single-vertex RBCs.*
- *For all vertices,* $left(V) \neq right(V)$.
- *If* $op(V) = \leftrightarrow$ *then* $left(V)$ *and* $right(V)$ *are unsigned.*
- *There is a total order* $\prec$ *such that for all* $V \in \mathbf{V}$, $left(V) \prec right(V)$.

For example, Figure 1a shows the RBC representing the formula $a \wedge b \leftrightarrow \neg(a \rightarrow b)$, with some internal vertices annotated by a subscript capital. The annotations allow us to refer to the subformula $a \wedge b$ by the vertex A, for example, and also allows us to depict RBC fragments by identifying a vertex without giving any further details.

To simplify the definitions in this paper we extend the set of properties on RBC vertices and edges with the inverse functions of *target*, and *left* and *right*:

$$inedges(V) = \{E | E \in \mathbf{E}, target(E) = V\}$$

$$source(E) = \begin{cases} V & \text{if } E = left(V) \vee E = right(V) \\ \text{undefined} & \text{otherwise} \end{cases}$$

RBC operations Two RBCs rooted with edges L and R, may be composed given an operation $o \in \{\wedge, \leftrightarrow\}$ and a sign $s \in \{+, -\}$ to give the RBC $rbc(L, R, o, s)$ as follows:

- If o may be trivially evaluated using identity and other properties, return the result of doing so.
- Otherwise, check $L \prec R$ and swap if not.
- If $o = \leftrightarrow$ then s becomes $s \oplus sign(L) \oplus sign(R)$, and $sign(L)$ and $sign(R)$ become $+$ ($\oplus$ is the exclusive-or operation).
- The new vertex $V = \langle o, L, R \rangle$ is inserted into the DAG.
- The result is the edge $\langle sign, V \rangle$.

3 CNF Conversions on Linear Trees

We begin by examining CNF conversions for a restriction of RBCs, which will become a building block for the CNF conversions of full RBCs. Linear trees represent linear formulæ (those without equivalence operators) without taking into account the possibility for sharing.

Definition 2 *A* linear tree *is an RBC with the following changes to its structure:*

- *The only internal vertices are conjunction vertices*
- *No vertices are shared: the graph is a tree*

Given a linear tree, we define the following additional properties over vertices

$$inedge(V) = E \qquad \text{where } target(E) = V$$

$$sib(V) = \begin{cases} target(left(V')) & \text{if } inedge(V) = right(V') \\ target(right(V')) & \text{if } inedge(V) = left(V') \end{cases}$$

We give the various well-known CNF conversions informally and as depth-first procedures on linear trees. Each conversion produces a set of clauses which may be treated using the union ($\cup$) operator, combining two sets of clauses together, and the cross-multiply operator ($\times$), which forms the set of clauses corresponding to the disjunction of two sets, obtained by

$$a \times b = \{x \cup y | x \in a, y \in b\}$$

We use the notation $|C|$ to refer to the number of clauses in set C.

The *standard* CNF conversion is that obtained by exploiting the distributive properties of $\wedge$ and $\vee$ on a formula already in NNF to push disjunctions in towards the literals. This produces an equivalent (rather than equisatisfiable) formula at the expense of a potentially exponential number of clauses. Nevertheless, the conversion is optimal for some input formulæ. We define the conversion for linear trees as a recursive descent. CNF(T) given in Figure 2 denotes the standard CNF conversion of the subtree beginning at a root edge T.

$$\mathrm{CNF}(E) = \begin{cases} \mathrm{CNF}(\mathit{target}(V)) & \text{if } \mathit{sign}(E) = + \\ \mathrm{CNF}^-(\mathit{target}(V)) & \text{if } \mathit{sign}(E) = - \end{cases}$$

$$\mathrm{CNF}^-(E) = \begin{cases} \mathrm{CNF}^-(\mathit{target}(V)) & \text{if } \mathit{sign}(E) = + \\ \mathrm{CNF}(\mathit{target}(V)) & \text{if } \mathit{sign}(E) = - \end{cases}$$

$$\mathrm{CNF}(V) = \begin{cases} \mathit{var}(V) & \text{if } V \in \mathbf{V_L} \\ \mathrm{CNF}(\mathit{left}(V)) \cup \mathrm{CNF}(\mathit{right}(V)) & \text{if } \mathit{op}(V) = \wedge \end{cases}$$

$$\mathrm{CNF}^-(V) = \begin{cases} \neg\, \mathit{var}(V) & \text{if } V \in \mathbf{V_L} \\ \mathrm{CNF}^-(\mathit{left}(V)) \times \mathrm{CNF}^-(\mathit{right}(V)) & \text{if } \mathit{op}(V) = \wedge \end{cases}$$

Fig. 2. The standard clause form conversion for linear trees

3.1 Clause Form Conversions with Renaming

Renaming subformulæ is a strategy for reducing the number of clauses produced by a formula. The observation is made that a subformula may be replaced by a single variable if clauses are given to constrain that variable such that the satisfiability of the overall formula is unaffected. Such a conversion is said to be *equisatisfiable*: the introduced variables break equivalency. For example, the formula $(a \wedge b \wedge c) \vee (d \wedge e \wedge f)$ produces nine clauses in the standard conversion; introducing a new variable for the left-hand disjunct to produce the formula

$$x_{a \wedge b \wedge c} \vee (d \wedge e \wedge f) \quad \wedge \quad x_{a \wedge b \wedge c} \leftrightarrow (a \wedge b \wedge c)$$

with $x_{a \wedge b \wedge c}$ constrained by the equivalence on the right hand side results in only seven clauses. Nevertheless, it is satisfiable by precisely those assignments that satisfy the original formula.

The most straightforward algorithm of this type gives a new name to every internal vertex of the tree and is known as the *definitional* clause form conversion, given in Figure 3.

In fact, as observed by Plaisted and Greenbaum [10], if a subformula occurs with positive or negative polarity — if it appears under an even or odd number of negations — then only an implication is required to constrain the new variable, with the direction of the implication corresponding to the polarity of the subformula. We define the polarity function $\mathrm{pol}(T, V)$ for a vertex V in a linear trees T as

$$\mathrm{pol}(T, T) = 1$$

$$\mathrm{pol}(T, E) = \begin{cases} \mathrm{pol}(T, \mathit{source}(E)) & \text{if } \mathit{sign}(E) = + \\ -\,\mathrm{pol}(T, \mathit{source}(E)) & \text{if } \mathit{sign}(E) = - \end{cases}$$

$$\mathrm{pol}(T, V) = \mathrm{pol}(\mathit{inedge}(V))$$

$$\mathrm{DEF}(E) = \begin{cases} \mathrm{DEF}(\mathit{target}(V)) & \text{if } \mathit{sign}(E) = + \\ \mathrm{DEF}^{-}(\mathit{target}(V)) & \text{if } \mathit{sign}(E) = - \end{cases}$$

$$\mathrm{DEF}(V) = \begin{cases} \mathit{var}(V) & \text{if } v \in \mathbf{V_L} \\ \{\{\neg x_V, x_{\mathit{target}(\mathit{left}(V))}\}, \{\neg x_V, x_{\mathit{target}(\mathit{right}(V))}\}\} & \\ \quad \cup \{\{x_V, \neg x_{\mathit{target}(\mathit{left}(V))}, \neg x_{\mathit{target}(\mathit{right}(V))}\}\} & \\ \quad \cup \mathrm{DEF}(\mathit{left}(V)) \cup \mathrm{DEF}(\mathit{right}(V)) & \text{if } \mathit{op}(V) = \wedge \end{cases}$$

$$\mathrm{DEF}^{-}(V) = \begin{cases} \neg\, \mathit{var}(V) & \text{if } v \in \mathbf{V_L} \\ \{\{x_V, x_{\mathit{target}(\mathit{left}(V))}\}, \{x_V, x_{\mathit{target}(\mathit{right}(V))}\}\} & \\ \quad \cup \{\{\neg x_V, \neg x_{\mathit{target}(\mathit{left}(V))}, \neg x_{\mathit{target}(\mathit{right}(V))}\}\} & \\ \quad \cup \mathrm{DEF}(\mathit{left}(V)) \cup \mathrm{DEF}(\mathit{right}(V)) & \text{if } \mathit{op}(V) = \wedge \end{cases}$$

Fig. 3. The definitional clause form conversion

In the example above, the subformula $a \wedge b \wedge c$ appears positively, so the renaming can be shortened to

$$x_{a \wedge b \wedge c} \vee (d \wedge e \wedge f) \quad \wedge \quad x_{a \wedge b \wedge c} \rightarrow (a \wedge b \wedge c)$$

producing only six clauses.

For linear trees, we consider only renamings of *vertices* (other analyses place an equivalent restriction on renaming subfomulæ with negation as the main connective). The order in which renamings are made does not affect the final result due to the commutivity of $\wedge$, so we are able to give renaming-based clause form conversions in terms of the sets of vertices that they rename. The transformation in Figure 4 constructs a graph consisting of the renamed formula and the subgraph defining constraints on the new variables. This is sufficient to allow us to write the structure-preserving clause form conversion due to Plaisted and Greenbaum [10] as

$$\mathrm{SP}(T) = \mathrm{CNF}(\mathrm{ren}(T, \mathbf{V_I}))$$

It is easy to construct cases where the definitional and structure-preserving conversions perform significantly worse than the standard conversion, despite the difference in asymptotic complexity. Consider, for example, the case of a formula already in conjunctive normal form. The structure-preserving conversion involves producing a new variable for each clause, with each definition taking one clause. The result is a worst-case doubling in the size of the clause form, where the standard conversion leaves the formula unchanged. A better approach is to construct the renaming sets more carefully, according to the overall impact that a renaming has.

3.2 The Conversion due to Boy de la Tour

Boy de la Tour [4] presents a comprehensive solution to the problem of choosing the subformulæ to rename. The approach taken is to compute the impact of

$$\mathrm{ren}(T,\mathbf{R}) = rbc(\mathrm{def}(T,T,\mathbf{R}),\mathrm{sub}(T,\mathbf{R}),\wedge,+)$$

$$\mathrm{def}(T,E,\mathbf{R}) = \mathrm{def}(T,target(E),\mathbf{R})$$

$$\mathrm{def}(T,V,\mathbf{R}) = \begin{cases} V & \text{if } V \in \mathbf{V_L} \\ rbc(\left\{\begin{array}{ll} \top & \text{if } V \notin \mathbf{R} \\ rbc(x_V,\mathrm{sub}^-(V,\mathbf{R}\setminus\{V\}),\wedge,-) & \text{if } \mathrm{pol}(T,V)=1 \\ rbc(\neg x_V,\mathrm{sub}^+(V,\mathbf{R}\setminus\{V\}),\wedge,-) & \text{if } \mathrm{pol}(T,V)=-1 \end{array}\right\}, & \\ \quad rbc(\mathrm{def}(T,left(V),\mathbf{R}),\mathrm{def}(T,right(V),\mathbf{R}),\wedge,+), & \\ \quad \wedge,+) & \text{if } V \in \mathbf{V_I} \end{cases}$$

$$\mathrm{sub}(E,\mathbf{R}) = \mathrm{sub}^{sign(E)}(target(E),\mathbf{R})$$

$$\mathrm{sub}^s(T,V,\mathbf{R}) = \begin{cases} V & \text{if } V \in \mathbf{V_L} \\ x_V & \text{if } V \in \mathbf{R} \\ rbc(\mathrm{sub}(left(V),\mathbf{R}),\mathrm{sub}(right(V),\mathbf{R}),op(V),s) & \text{otherwise} \end{cases}$$

Fig. 4. The vertex-based renaming construction $\mathrm{ren}(T,\mathbf{R})$. Function $\mathrm{sub}(T,\mathbf{R})$ returns the graph with root edge T with renamed subgraphs replaced by variables; $\mathrm{def}(T,T',\mathbf{R})$ returns the graph defining all the introduced variables below T' with respect to root T

Table 1. The clause counting functions $p^+(V)$ and $p^-(V)$

	$p^+(E)$	$p^-(E)$
$sign(E)=+$	$p^+(target(E))$	$p^-(target(E))$
$sign(E)=-$	$p^-(target(E))$	$p^+(target(E))$
	$p^+(V)$	$p^-(V)$
$v \in \mathbf{V_L}$	1	1
$op(V)=\wedge$	$p^+(left(V))+p^+(right(V))$	$p^-(left(V))p^-(right(V))$

renaming any given subformula and to perform the renaming only if it will not increase the number of clauses produced by the formula as a whole. The conversion is shown to be optimal for formulæ without equivalences, and we will make use of this property in order to prove the optimality of the new conversion in Section 4.

Boy de la Tour defines the functions $p^+(T) = |\mathrm{CNF}(T)|$ and $p^-(T) = |\mathrm{CNF}(\neg T)|$ using a simple lookup table (Table 1) which enables these values to be computed without constructing the clauses themselves. The *benefit* (that is, the reduction in the total number of clauses) of renaming a vertex V in a tree T is given by

$$B(T,V) = p^+(T) - p^+(\mathrm{ren}(T,\{V\}))$$

In order to make a decision about renaming at a particular vertex without needing to analyse the whole tree, $p^+(T)$ is rewritten in terms of $p^+(V)$ and $p^-(V)$:

$$p^+(T) = a_V^T p^+(V) + b_V^T p^-(V) + c_V^T$$

Table 2. Computation of the coefficients a_V^T and b_V^T

	a_E^T	b_E^T
$E = T$	1	0
$sign(E) = +$	$a^T_{source(E)}$	$b^T_{source(E)}$
$sign(E) = -$	$b^T_{source(E)}$	$a^T_{source(E)}$
	a_V^T	b_V^T
$op(V) = \wedge$	$a^T_{inedge(V)}$	$b^T_{inedge(V)} p^-(sib\, V)$

$$\mathrm{BDLT}(T, E) = \mathrm{BDLT}(T, \mathit{target}(V))$$

$$\mathrm{BDLT}^+(T, V) = \begin{cases} \emptyset & \text{if } v \in \mathbf{V_L}, \text{ or} \\ \mathrm{BDLT}(T, \mathit{left}(V)) \cup \mathrm{BDLT}^+(T, \mathit{right}(V)) & \text{if } B(T, V) < 0, \text{ or} \\ \{V\} \cup \mathrm{BDLT}(\mathrm{ren}(T, \{V\}), \mathit{left}(v)) & \\ \quad \cup \mathrm{BDLT}(\mathrm{ren}(T, \{V\}), \mathit{right}(V)) & \text{if } B(T, V) \geq 0 \end{cases}$$

Fig. 5. Renaming sets construction for the Boy de la Tour conversion

Where the coefficients a, b may be considered as the number of occurrences of the clauses representing V and $\neg V$ respectively, such that the first sum counts the total number of clauses including subformulæ of V; the coefficient c represents the number of clauses due to the rest of the tree. a and b are computed from the context of V as in Table 2. Note that the values are related to the polarity of the vertices: $a_V^T = 0$ if $\mathrm{pol}(T, V) = -1$ and $b_V^T = 0$ if $\mathrm{pol}(T, V) = 1$. When computing the benefit, the coefficient c is cancelled, so we do not need to give its construction. The benefit function can now be given in terms of polarity as

$$\begin{aligned} a_V^T p^+(V) \; - \; (a_V^T + p^+(V)) \qquad & \text{if } \mathrm{pol}(T, V) = 1 \\ b_V^T p^-(V) \; - \; (b_V^T + p^-(V)) \qquad & \text{if } \mathrm{pol}(T, V) = -1 \end{aligned}$$

The algorithm given by Boy de la Tour is a top-down computation of the benefit of a renaming given the renamings that have gone before. We give the construction of the renaming set in Table 5 allowing us to write the algorithm as

$$\mathrm{BDLT}(T) = \mathrm{CNF}(\mathrm{ren}(T, \mathrm{BDLT}^+(T, T) \cup \mathrm{BDLT}^-(T, T)))$$

A dynamic programming implementation of $B(T, V)$ as given by Boy de la Tour [4] requires $O(1)$ computations at each vertex but the arithmetic is on $|\mathbf{V}|$-bit words which leads to a per-vertex complexity of $O(|\mathbf{V}|)$. The resulting algorithm is $O(|\mathbf{V}|^2)$ in contrast to DEF and SP which are both linear.

A more recent presentation of the algorithm by Nonnengart et al. [9] removes the requirement for arbitrary-length arithmetic by reducing $B(T, V) \geq 0$ to a number of case splits. Unfortunately, these can become quite elaborate: the conditions for zero polarity formulæ require the evaluation of eight syntactic conditions in various combinations.

	$p_r^+(E,\mathbf{R})$	$p_r^-(E,\mathbf{R})$
$sign(E)=+$	$p_r^+(target(E),\mathbf{R})$	$p_r^-(target(E),\mathbf{R})$
$sign(E)=-$	$p_r^-(target(E),\mathbf{R})$	$p_r^+(target(E),\mathbf{R})$

	$p_r^+(V,\mathbf{R})$	$p_r^-(V,\mathbf{R})$
$V\in\mathbf{V_L}$	1	1
$V\in\mathbf{R}$	1	1
$op(V)=\wedge$	$p_r^+(left(V),\mathbf{R})+p_r^+(right(V),\mathbf{R})$	$p_r^-(left(V),\mathbf{R})p_r^-(right(V),\mathbf{R})$

Table 3. The renaming-compensated clause counting functions $p_r^+(T,\mathbf{R})$ and $p_r^-(T,\mathbf{R})$

$$\mathrm{COMP}(T,E)=\mathrm{COMP}(T,target(V))$$

$$\mathrm{COMP}(T,V)=\begin{cases}\emptyset & \text{if } V\in\mathbf{V_L}, \text{ or}\\ \mathrm{COMP}(T,left(V))\cup\mathrm{COMP}(T,right(V)) & \text{if } \mathrm{pol}(T,V)=1, \text{ or}\\ \mathrm{dis}(V)\cup\mathrm{COMP}^-(T,left(V))\cup\mathrm{COMP}^-(T,right(V)) & \text{if } \mathrm{pol}(T,V)=-1\end{cases}$$

$$\mathrm{dis}(V)=\left\{\begin{array}{ll}\emptyset & \text{if } n_ln_r\le n_l+n_r, \text{ or}\\ \{left(V)\} & \text{if } n_l>n_r\\ \{right(V)\} & \text{if } n_l\le n_r\end{array}\right\} \text{ where } \left\{\begin{array}{l}n_l=p_r^-(left(V),\mathrm{COMP}(T,left(V)))\\ n_r=p_r^-(right(V),\mathrm{COMP}(T,right(V)))\end{array}\right\}$$

Fig. 6. Renaming sets construction for the compact conversion

4 The Compact Conversion

We present a new clause form conversion, the *compact* conversion which computes the sets of renaming locally and bottom-up. For each vertex we consider the number of clauses it will generate based on whether a child vertex is renamed. Consider a disjunction $\phi\vee\psi$, with all subformulæ of ϕ and ψ already renamed as appropriate. The disjunction is converted by either renaming an argument, eg ϕ to x_ϕ, which produces a definition $x_\phi\rightarrow\phi$ and replaces the disjunction by the renamed form $x_\phi\vee\psi$; or alternatively computing $\mathrm{CNF}(\phi)\times\mathrm{CNF}(\psi)$ — the standard conversion of the disjunction. The decision is made based on which generates the most clauses, determined by the sum or the product, respectively, of the number of clauses in ϕ and ψ.

More precisely, we define the function $\mathrm{COMP}(T,V)$ in Figure 6 to give the set of renamings on the tree beginning at V. The auxiliary function $\mathrm{dis}(V)$ chooses the best child of V, if any, to rename by using the sum-versus-product decision. The renaming condition is computed on the tree after all vertices below the considered one have been renamed. To accommodate this we define a new pair of clause-counting functions $p_r^+(V,\mathbf{R})$ and $p_r^-(V,\mathbf{R})$ which count the number of clauses produced by the graph beginning at vertex V after the application of renaming $\mathbf{R}$ (Table 3). That is, $p_r^s(V,\mathbf{R})=|\mathrm{sub}^s(V,\mathbf{R})|$ (the clauses in $\mathrm{def}^s(V,\mathbf{R})$ are disregarded as they play no further part in determining the size of the result).

Since we are targeting a SAT solver with this conversion, with its (assumed) exponential complexity in the number of variables, we choose to rename only if it *reduces* the number of clauses produced. In the case that the number of clauses is the same, the renaming is not performed. This is in contrast to the Boy de la Tour conversion, where the optimality analysis is simplified by the zero-benefit renaming.

5 Optimality of the Compact Conversion for Linear Trees

We show the optimality of the compact conversion by a comparison with the Boy de la Tour conversion. We establish which vertices appear in the renaming sets of one conversion and not the other, and then analyse the impact that the differences make.

When comparing the decision taken to include a vertex in the renaming sets by the two algorithms we take into account the different contexts: in the Boy de la Tour algorithm, the superformulæ have already been renamed; in the compact conversion the subformulæ have been renamed. Writing $\mathbf{R}$ for a set of renamings, we have $\mathbf{R}_{\sqsupset V}$ for the subset of renamings involving the superformulæ of V and $\mathbf{R}_{\sqsubset V}$ for the subset involving the subformulæ of V. The compact conversion depends only on p_r^+ and p_r^- but these are computed after subformula renaming. That is, the decision to rename the vertex V_1 in $V_1 \wedge V_2$ is based on the values $p_r^+(V_1, \mathbf{R}_{\sqsubset V_1})$, $p_r^-(V_2, \mathbf{R}_{\sqsubset V_2})$ and their complements. In contrast, for the Boy de la Tour algorithm the decision is based on the values $a_{V_1}^{\mathrm{ren}(T,\mathbf{R}_{\sqsupset V_1})}$, $b_{V_1}^{\mathrm{ren}(T,\mathbf{R}_{\sqsupset V_1})}$, $p^+(V_1)$, $p^-(V_1)$

We begin by establishing some basic lemmas about the Boy de la Tour coefficients and the clause counting functions.

Lemma 1. *For a vertex V and renaming $\mathbf{R}$ on tree T, $a_V^{\mathrm{ren}(T,\mathbf{R})} = 1$ if $\mathrm{pol}(T, V) = 1$, and $b_V^{\mathrm{ren}(T,\mathbf{R})} = 1$ if $\mathrm{pol}(T, V) = -1$*

Proof. After renaming, a vertex V becomes part of the definition of the replacement variable x_V. According to Figure 4, the definition is attached by a tree of positive conjunctions to the root with the sign of the inedge of V reflecting its original polarity. By the definition of a_V^T and b_V^T on conjunctions, the lemma holds.

Lemma 2. *For a vertex V and renamings $\mathbf{R}$ and $\mathbf{R}'$ with $\mathbf{R}' \subseteq \mathbf{R}$, $p_r^s(V, \mathbf{R}) \leq p_r^s(V, \mathbf{R}') \leq p^s(V)$*

Proof. This follows from the definitions of p_r^s and p_r. Both increase monotonically with tree depth. As renaming effectively prunes part of the tree, it can only reduce the values of the functions.

5.1 Positive Polarity

Lemma 3. *Neither conversion renames the children of positive polarity conjunctions. That is, for $\mathbf{pc} = \{V \in \mathbf{V}_I \mid \mathrm{pol}(T, \mathit{source}(\mathit{inedge}(V))) = 1\}$, $\mathbf{pc} \cap \mathrm{BDLT}(T, V) = \emptyset$ and $\mathbf{pc} \cap \mathrm{COMP}(T, V) = \emptyset$*

Proof. The argument for the compact conversion follows trivially from its definition. For the Boy de la Tour conversion, consider the vertex X in Figure 7a. The benefit of renaming, $B(T, X)$, is evaluated in the context $\text{ren}(T, \mathbf{R}_{\sqsupset X})$. From Figure 2, $a_X^{\text{ren}(T,\mathbf{R}_{\sqsupset X})} = a_B^{\text{ren}(T,\mathbf{R}_{\sqsupset X})}$, hence the benefit is

$$a_B^{\text{ren}(T,\mathbf{R}_{\sqsupset X})} p^+(X) \; - \; (a_B^{\text{ren}(T,\mathbf{R}_{\sqsupset X})} + p^+(X))$$

The condition $B(T, X) \geq 0$ reduces to $a_B^{\text{ren}(T,\mathbf{R}_{\sqsupset X})} \geq 2$ and $p^+(X) \geq 2$. From Lemma 1, in order to obtain the former vertex B must not be renamed. From $B \notin \mathbf{R}$, we deduce $\mathbf{R}_{\sqsupset B} = \mathbf{R}_{\sqsupset X}$ and hence write the condition $B(T, B) < 0$ as

$$a_B^{\text{ren}(T,\mathbf{R}_{\sqsupset X})} p^+(B) \; - \; (a_B^{\text{ren}(T,\mathbf{R}_{\sqsupset X})} + p^+(B)) < 0$$

which together with the earlier conditions constrains $p^+(B) = 1$. Since B is a conjunction it produces $p^+(X) + p^+(Y)$ clauses and the condition on $p^+(X)$ is thus in conflict with the condition that B is not renamed.

The argument for Y follows similarly, as does the case of BX or BY being signed edges.

5.2 Negative Polarity

We break the negative polarity argument into several pieces, firstly simplifying the Boy de la Tour benefit function. Consider vertex X in Figure 7b. From Figure 2, $b_X^{\text{ren}(T,\mathbf{R}_{\sqsupset X})} = b_B^{\text{ren}(T,\mathbf{R}_{\sqsupset X})} p^-(Y)$, hence the benefit of renaming $B(T, X)$, in the context $\text{ren}(T, \mathbf{R}_{\sqsupset X})$ is

$$b_B^{\text{ren}(T,\mathbf{R}_{\sqsupset X})} p^-(Y) p^+(X) \; - \; (b_B^{\text{ren}(T,\mathbf{R}_{\sqsupset X})} p^-(Y) + p^+(X))$$

We consider two cases for $B(T, X) \geq 0$. If $b_B^{\text{ren}(T,\mathbf{R}_{\sqsupset X})} = 1$ then the renaming decision is localised: it is based only on $p^+(X)$ and $p^-(Y)$:

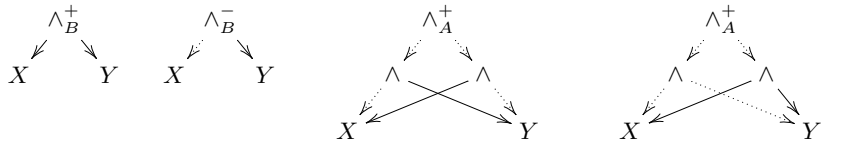
$$B'(T, X) = p^-(Y) p^+(X) - (p^-(Y) + p^+(X))$$

If $b_B^{\text{ren}(T,\mathbf{R}_{\sqsupset X})} \geq 2$, we must consider the same situation as for the positive case: the condition that $B \notin \mathbf{R}^-$. The inequality $B(T, B) < 0$ reduces to

$$b_B^{\text{ren}(T,\mathbf{R}_{\sqsupset X})} p^-(B) < b_B^{\text{ren}(T,\mathbf{R}_{\sqsupset X}))} + p^-(B)$$

This holds only when $p^-(B) = 1$. Given $p^-(B) = p^+(X) p^-(Y)$ we also have $p^+(X) = p^-(Y) = 1$ and hence the vertex X is not renamed. This configuration is covered by the reduced condition $B'(T, X)$ which is thus sufficient condition for making the renaming decision. That is, the renaming decision is made independently of the value of $b_B^{\text{ren}(T,\mathbf{R}_{\sqsupset X})}$.

Lemma 4. *For linear trees, the renaming given by the Boy de la Tour algorithm with benefit function $B'(T, V)$ is the same as with the original function $B(T, V)$.*



(a) Positive (b) Negative (c) Positive equivalence (d) Negative equivalence

Fig. 7. RBC subgraphs for the optimality proofs and equivalence discussion

Proof. The argument for the children of negative polarity vertices is given above (the arguments for Y and different edge signs follow similarly). For children of positive polarity vertices, it is easy to see that Lemma 3 still holds. The remaining case is the root vertex, which is not renamed under either condition.

Using this reduced condition, the Boy de la Tour conversion has no restriction on the order of evaluation, which means that we can compare it more directly with the compact conversion. We define the conversion $\text{BDLT}'(T, V)$ to be a bottom-up conversion using the benefit function $B'(T, V)$. From Lemmas 3 and 4 we know that $\text{BDLT}(T, T) = \text{BDLT}'(T, T)$ for all linear trees T. All remaining theorems are on this bottom-up conversion.

Lemma 5. *For all linear trees* T, $\text{COMP}(T, T) \subseteq \text{BDLT}'(T, T)$

Proof. We argue in the negative as it is more convenient. Consider vertex X in Figure 7a, with $X \notin \text{BDLT}'(T, T)$. From the definition of the Boy de la Tour conversion, $B'(T, X) < 0$ which reduces to the two possibilities $p(X) = 1$ or $p(Y) = 1$. By Lemma 2, this means that either $p_r^+(X, \mathbf{R}_{\sqsubset B}) = 1$ or $p_r^+(Y, \mathbf{R}_{\sqsubset B}) = 1$ and hence the renaming condition for the compact conversion, $p_r^+(X, \mathbf{R}_{\sqsubset B})p_r^+(Y, \mathbf{R}_{\sqsubset B}) > p_r^+(X, \mathbf{R}_{\sqsubset B}) + p_r^+(Y, \mathbf{R}_{\sqsubset B})$, is violated.

The argument follows similarly for Y.

Lemma 6. *For all linear trees* T, *with a renaming* $\mathbf{R} = \text{COMP}(T, T)$, *for all* $V \notin \mathbf{R}$, $p_r^s(V, \mathbf{R}) = 1 \rightarrow p^s(V) = 1$

Proof. We show this by induction on the structure of the tree. The base case $V \in \mathbf{V_L}$ (V is a leaf) is trivial from the definition of p. For the step case, if V is a disjunction, then $p_r^s(\mathit{left}(V), \mathbf{R}) = p_r^s(\mathit{right}(V), \mathbf{R}) = 1$. This means, if $X = \mathit{target}(\mathit{left}(V))$ and $Y = \mathit{target}(\mathit{right}(V))$,

- $X \notin \mathbf{R}, Y \notin \mathbf{R}$: proof follows from the inductive hypothesis
- $X \notin \mathbf{R}, Y \in \mathbf{R}$: the condition necessary to rename Y is violated because, by Lemma 2, $p_r^s(X, \mathbf{R}_{\sqsubset V}) = p^s(X) = 1$.
- $X \in \mathbf{R}, Y \notin \mathbf{R}$: as above, by symmetry
- $X \in \mathbf{R}, Y \in \mathbf{R}$: prohibited by the definition of the compact conversion

V cannot be a conjunction as $p_r^s(V, \mathbf{R}) \geq 2$ is in contradiction with the induction hypothesis.

We can now fix the precise difference between the two conversions. Consider vertex X in Figure 7a, with $X \notin \text{COMP}(T,T)$. By the definition of the compact conversion, $p_r^+(X, \mathbf{R}_{\sqsubset B})p_r^+(Y, \mathbf{R}_{\sqsubset B}) \leq p_r^+(X, \mathbf{R}_{\sqsubset B}) + p_r^+(Y, \mathbf{R}_{\sqsubset B})$ which reduces to the three possibilities $p_r^+(X, \mathbf{R}_{\sqsubset B}) = 1$ or $p_r^+(Y, \mathbf{R}_{\sqsubset B}) = 1$ or $p_r^+(X, \mathbf{R}_{\sqsubset B}) = p_r^+(Y, \mathbf{R}_{\sqsubset B}) = 2$. In the first case, X may be a leaf vertex, in which case $X \notin \text{BDLT}'(T,T)$, or a disjunction, in which case by Lemma 6, $p^+(X) = 1$ and hence[1] $X \notin \text{BDLT}'(T,T)$. A conjunction is ruled out by the restriction on the number of clauses. The cases for Y and for signed edges follow similarly. For the final case, by Lemma 2, the Boy de la Tour conversion always renames either X or Y: this defines the set of vertices renamed by Boy de la Tour but not by compact.

Lemma 7. *For all linear trees T, $\text{COMP}(T,T) \cup \mathbf{Z} = \text{BDLT}'(T,T)$ where $\mathbf{Z}$ is the set of vertices such that for all $V \in \mathbf{Z}$, $p_r^+(V, \text{COMP}(T,V)) = 2$ and $p_r^+(sib(V), \text{COMP}(T,V)) = 2$*

Proof. From the discussion above and Lemma 5, no other vertex is in $\text{BDLT}'(T,T)$ that is not in $\text{COMP}(T,T)$.

Theorem 1. *The size of the clause form generated by the compact and Boy de la Tour conversions is the same: $p_r^+(T, \text{COMP}(T,T)) = p_r^+(T, \text{COMP}(T,T))$*

Proof. Since renamings may be applied in any order, we show that after applying those in $\text{COMP}(T,T)$, the benefit of applying any of those in $\mathbf{Z}$ is zero. By Boy de la Tour's *fundamental theorem of monotonicity* [4], the members of $\mathbf{Z}$ may be considered in any order for this proof.

Consider a vertex $X \in \mathbf{Z}$ as depicted in Figure 7b. The benefit $B'(T,X)$ of renaming X after $\text{COMP}(T,T)$ is $p_r^+(X, \text{COMP}(T,T))p_r^-(Y, \text{COMP}(T,T)) - (p_r^+(X, \text{COMP}(T,T)) + p_r^-(Y, \text{COMP}(T,T))$. However, by the definition of $\mathbf{Z}$ in Lemma 7, and by Lemma 2, $p_r^+(X, \text{COMP}(T,T)) = 2$ and $p_r^+(Y, \text{COMP}(T,T)) = 2$, and hence $B'(T,V) = 0$.

6 Extension to RBCs

We have shown that the compact conversion produces an optimal number of clauses for linear trees, so we now extend the algorithm to general RBCs. The extension is heuristic: like Boy de la Tour, we do not claim optimality for the resulting clause form conversion.

Removal of Equivalences An RBC with equivalence vertices can be transformed into a linear RBC with only a linear increase in size by replacing equivalences with the subgraphs given in Figures 7c and d. The different treatments for positive and negative polarity equivalences reduce the number of clauses generated [10]. Note that a negative equivalence is replaced by a positive subgraph so the incoming edge must have its sign inverted.

[1] The case split for BDLT′ is given in the proof of Lemma 5

Polarity Zero Vertices The children of equivalence nodes are referenced both positively and negatively (as can be seen from the replacement subgraphs), sometimes referred to as *zero* polarity. Similarly, the sharing used in RBCs encourages a single vertex to be referenced with both polarities. We can convert an RBC with zero polarity vertices to one without by splitting every zero polarity vertex into a pair, one of each polarity, and suitably treating the incoming edges. Such treatment results in at most a doubling of the size of the RBC.

The substitution and subsequent splitting of equivalences differs significantly from the direct treatment of Boy de la Tour. In particular, Boy de la Tour's algorithm renames a descendant vertex of an equivalence both positively and negatively, simultaneously. This sometimes results in a tradeoff: the renaming of one polarity must have sufficient benefit to outweigh any negative benefit of renaming the other polarity. By splitting the polarities and treating them independently we improve the flexibility of the conversion and reduce the number of clauses in some circumstances, as compared to Boy de la Tour.

Shared Subgraphs Having removed equivalences and zero polarity vertices we are close to a linear tree structure. In fact, we can see the resulting structure as a collection of trees joined at the shared vertices. We can incorporate treatment of shared vertices into the bottom-up compact conversion algorithm by renaming any shared vertex which generates more than one clause and repeating the subgraph otherwise. The resulting algorithm is locally optimal as each constituent tree is optimally converted and the shared subgraphs are renamed only when renaming does not increase the resulting size.

7 Implementation and Evaluation

We have implemented the compact conversion extended to RBCs as part of the NuSMV model checker [5]. The implementation works directly on RBCs, performing the substitutions and duplications described above implicitly rather than constructing the resulting graph explicitly. Each vertex is considered as both a positive and a negative polarity vertex, and a depth-first traversal is used to mark each vertex with the number of incoming edges in each polarity. A second depth-first traversal produces the clause form directly. Bottom-up, each vertex is annotated with the clauses produced positively and negatively after renaming (ie, $\mathrm{CNF}(\mathrm{sub}(V, \mathrm{COMP}(T, V)))$), the definitional clauses being saved in a global variable (ie, $\mathrm{CNF}(\mathrm{def}(V, \mathrm{COMP}(T, V)))$). Whenever a shared vertex is encountered, it is renamed according to the strategy described above. No explicit computation of $p_r^s(V)$ is required: they correspond to the sizes of the sets of clauses — a constant time operation.

In Table 4 we compare the behaviour of the built-in CNF conversion in NuSMV (the definitional conversion) against the structure-preserving conversion and the compact conversion using two leading satisfiability solvers. The

Table 4. Benchmark results for three clause form conversions

Problem	Conv.	Clauses	Vars	Total literals	zChaff [7] Decisions	zChaff [7] Time (s)	Jerusat [8] Time (s)
DME (Access)	Def	89150	31328	229882	40332	24.2	155.3
	SP	53285	22866	129840	39283	25.8	104.9
	Comp	22979	4986	70278	48232	10.6	32.1
DME (Priority)	Def	234515	79577	569387	28798	52.5	149.8
	SP	109637	51965	273339	21894	8.1	47.1
	Comp	52312	7587	456576	34936	5.2	3.53
DME (OT)	Def	737157	247079	1741365	25991	181.3	1084
	SP	280979	140302	700484	32023	50.4	150.9
	Comp	141604	12779	3322302	34808	10.4	38.9
Elevator	Def	234397	78483	548461	52450	68.3	369.2
	SP	109677	39373	274751	147791	74.4	338.3
	Comp	83901	23157	343673	168902	190	15.1

problems used are the standard DME benchmark[2] and a deadlock problem[3] (Elevator), as these were found to be representative of the behaviour on other hardware and deadlock problems. Unsurprisingly, the compact conversion consistently generates fewer clauses and the solving times are also better in most cases, sometimes dramatically so. More surprisingly, perhaps, is the increase in the number of decisions made by zChaff in every case: for the DME example, decisions are made more quickly, while for the Elevator, the rise in the number of decisions is more dramatic and the time taken for zChaff is increased. Interestingly, the time taken by Jerusat in this case is dramatically better than the best case for zChaff; it is otherwise usually outperformed by zChaff.

The results also illustrate the effect of the compact conversion preferring to repeat small sets of clauses rather than renaming them: the total number of literals is, in the worst case, double that for the definitional conversion; this is contrasted with the order of magnitude reductions in the number of variables!

8 Conclusions

Despite optimising a problem attribute that is not directly connected to the solving time — the number of clauses — the compact conversion algorithm produces a set of clauses that are in most cases more quickly solved. With the compact conversion, in contrast to the Boy de la Tour conversion, this is achieved without changing the complexity class of the conversion as compared to the more well-known clause form conversions.

[2] See [6] for more details

[3] Thanks to Toni Jussila for providing the files for this example

References

1. Parosh Aziz Abdulla, Per Bjesse, and Niklas Eén. Symbolic reachability analysis based on SAT-solvers. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 6th International Conference, TACAS'00*, volume 1785 of *Lecture Notes in Computer Science*, pages 411–425. Springer-Verlag, March 2000.
2. Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of Boolean cardinality constraints. In *Principles and Practice of Constraint Programming — 9th International Conference, CP 2003*, Lecture Notes in Computer Science, 2003.
3. Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In W.R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems. 5th International Conference, TACAS'99*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, July 1999.
4. Thierry Boy de la Tour. An optimality result for clause form translation. *Journal of Symbolic Computation*, 14:283–301, 1992.
5. A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Proceedings of the Eleventh Conference on Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 495–499, Trento, Italy, July 1999. Springer-Verlag.
6. Alan Frisch, Daniel Sheridan, and Toby Walsh. A fixpoint based encoding for bounded model checking. In M D Aagaard and J W O'Leary, editors, *Formal Methods in Computer-Aided Design; 4th International Conference, FMCAD 2002*, volume 2517 of *Lecture Notes in Computer Science*, pages 238–254, Portland, OR, USA, November 2002. Springer-Verlag.
7. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *39th Design Automation Conference*, pages 530–535, Las Vegas, June 2001.
8. Alexander Nadel. Backtrack search algorithms for propositional logic satisfiability: Review and innovations. Master's thesis, Tel-Aviv University, November 2002.
9. Andreas Nonnengart, Georg Rock, and Christoph Weidenbach. On generating small clause normal forms. In Claude Kirchner and Hélène Kirchner, editors, *Fifteenth International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 397–411. Springer-Verlag, 1998.
10. David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, September 1986.
11. M. N. Velev. Efficient translation of Boolean formulas to CNF in formal verification of microprocessors. In *Asia and South Pacific Design Automation Convference (ASP-DAC '04)*, January 2004.

BMC 2004 Preliminary Version

Bounded Model Checking with SNF, Alternating Automata, and Büchi Automata

Daniel Sheridan [1]

School of Informatics
University of Edinburgh
Edinburgh, UK

Abstract

Model checking of LTL formulæ is traditionally carried out by a conversion to Büchi automata, and there is therefore a large body of research in this area including some recent studies on the use of alternating automata as an intermediate representation.

Bounded model checking has until recently been apart from this, typically using a direct conversion from LTL to propositional logic. In this paper we give a new bounded model checking encoding using alternating automata and focus on the relationship between alternating automata and SNF. We also explore the differences in the way SNF, alternating, and Büchi automata are used from both a theoretical and an experimental perspective.

Key words: Bounded model checking, SNF, LTL, Büchi automata, Alternating automata

1 Introduction

Before the introduction of bounded model checking in 1999 [1], LTL model checking was typically performed by converting the formula to an automaton expressing the formula, forming the product with the model automaton, then checking the result for emptiness. Research into producing the smallest automaton for a given LTL formula has been extensive and varied. There is literature giving improvements to the original "GPVW" conversion algorithm [12] including simplifying the LTL before conversion, and the automaton after conversion (eg, [7]) as well as the conversion itself. Some recent work [10,11] proposes the use of alternating automata (AA) as an intermediate representation of the formula. The LTL to AA conversion is linear space so allows for simplifications to be easily performed before the exponential space conversion to a Büchi automaton.

[1] Email: d.j.sheridan@sms.ed.ac.uk

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Bounded model checking (BMC) has traditionally taken a different approach: the original paper [1] gives an encoding from LTL directly to propositional logic. Being defined recursively on the structure of the formula this (naïvely) appears to be exponential size in the number of states, although with careful treatment [5] the result is polynomial size. An alternative encoding [9] based directly on the fixpoint characterisations of LTL operators produces an encoding which is quadratic size, but may with care be reduced to linear size in the number of states. The use of LTL to automata conversions as part of bounded model checking was first explicitly suggested by de Moura et al. [6]. The only experimental comparison [5] is very brief and mainly exercises the LTL simplification available in many automata conversion programs.

Although there are grounds for distinguishing between the direct-to-propositional conversion and the conversions via automata as "syntactic" versus "semantic" [5], we demonstrate in this paper the close correspondence between SNF and alternating automata and their conversion procedures from LTL. We review the use of Büchi automata for BMC and give a new encoding to enable direct use of alternating automata. This allows us to compare more closely the use of the SNF encoding with the use of automata, to explore the advantages and disadvantages of each approach. We demonstrate some of these differences with a series of experiments.

2 Background

2.1 Bounded model checking

BMC solves the LTL model checking problem by observing a restricted number of states, k. Infinite counterexamples may be represented by a path of the form ab^ω: a *k-l-loop path* with $k = |ab|$ and $l = |a|$. We constrain a finite sequence of states π to be a k-l-loop by the assertion ${}_lL_k \doteq (\pi(k) = \pi(l))$ [2]. Alternatively we can give finite counterexamples as a *k-prefix path* for some LTL properties. In particular, it is not possible to show to give a counterexample for $\mathbf{F}\, f$ for a k-bounded path. Typically, we verify a model by examining a sequence of k states π interpreted as either a prefix or a loop; we write a disjunction over the k possible interpretations, testing all of the options for the type of path and the value of l simultaneously.

2.2 The Separated Normal Form

SNF [8] is a clause-like normal form based on the Separation Theorem of Gabbay, with the general form $\mathbf{G} \bigwedge_i (P_i \rightarrow F_i)$ where $P_i \rightarrow F_i$, called *rules* are restricted to (writing p and f for propositional formulæ)

[2] Note that we give an equivalence between $\pi(k)$ and $\pi(l)$ rather than the transition as used in the original presentation [1]

$$\textsc{Snf}_{[\mathbf{X}]}(\{\varphi \rightarrow \psi(\mathbf{X}\ f)\} \cup \Gamma) \doteq \left\{ \begin{array}{c} \varphi \rightarrow \psi(\underline{\mathbf{X}\ f}) \\ \underline{\mathbf{X}\ f} \rightarrow \mathbf{X}\ f \end{array} \right\} \cup \Gamma$$

$$\textsc{Snf}_{[\mathbf{F}]}(\{\varphi \rightarrow \psi(\mathbf{F}\ f)\} \cup \Gamma) \doteq \left\{ \begin{array}{c} \varphi \rightarrow \psi(\underline{\mathbf{F}\ f}) \\ \underline{\mathbf{F}\ f} \rightarrow \mathbf{F}\ f \end{array} \right\} \cup \Gamma$$

$$\textsc{Snf}_{[\mathbf{G}]}(\{\varphi \rightarrow \psi(\mathbf{G}\ f)\} \cup \Gamma) \doteq \left\{ \begin{array}{c} \varphi \rightarrow \psi(f \wedge \underline{\mathbf{X}\,\mathbf{G}\ f}) \\ \underline{\mathbf{X}\,\mathbf{G}\ f} \rightarrow \mathbf{X}\,(f \wedge \underline{\mathbf{X}\,\mathbf{G}\ f}) \end{array} \right\} \cup \Gamma$$

$$\textsc{Snf}_{[\mathbf{U}]}(\{\varphi \rightarrow \psi(f\ \mathbf{U}\ g)\} \cup \Gamma) \doteq \left\{ \begin{array}{c} \varphi \rightarrow \psi(g \vee (f \wedge \underline{\mathbf{X}(f\ \mathbf{U}\ g)})) \\ \underline{\mathbf{X}(f\ \mathbf{U}\ g)} \rightarrow \mathbf{X}\,(g \vee (f \wedge \underline{\mathbf{X}(f\ \mathbf{U}\ g)})) \\ \varphi \rightarrow \mathbf{F}\ g \end{array} \right\} \cup \Gamma$$

$$\textsc{Snf}_{[\mathbf{R}]}(\{\varphi \rightarrow \psi(f\ \mathbf{R}\ g)\} \cup \Gamma) \doteq \left\{ \begin{array}{c} \varphi \rightarrow \psi(g \wedge (f \vee \underline{\mathbf{X}(f\ \mathbf{R}\ g)})) \\ \underline{\mathbf{X}(f\ \mathbf{R}\ g)} \rightarrow \mathbf{X}\,(g \wedge (f \vee \underline{\mathbf{X}(f\ \mathbf{R}\ g)})) \end{array} \right\} \cup \Gamma$$

Fig. 1. The transformation function for SNF.

Initial rules of the form $\mathbf{start} \rightarrow f$ where **start** holds only in the initial state of each path

Global invariant rules $p \rightarrow f$ with no temporal operator

Global step rules $p \rightarrow \mathbf{X}\,f$

Global eventuality rules $p \rightarrow \mathbf{F}\,f$

Transformation from an LTL formula in NNF f to a set of SNF rules is achieved by repeatedly applying the transformation functions in Figure 1 to the initial formula set $\{\mathbf{start} \rightarrow f\}$ [9]. The transformations introduce new variables identified by the syntax $\underline{x}$ with x indicating the intuitive meaning of the variable. We write Γ for the subset of formulæ which are not affected by the transformation, φ and ψ for arbitrary LTL formulæ in NNF, and f and g for propositional formulæ. We also write $\psi(\mathbf{G}\,f)$ to say that $\mathbf{G}\,f$ occurs in ψ, while $\psi(g)$ stands for the formula obtained by substituting every occurrence of $\mathbf{G}\,f$ with g in ψ; similarly for the other temporal operators.

2.3 Büchi Automata

We cover Büchi automata only briefly here; we direct the interested reader to the tutorial paper by Wolper [17].

Definition 2.1 A Büchi automaton $\mathcal{B}$ is defined by the tuple $\langle Q, \Sigma, \delta, I, T \rangle$ where Q is the set of states; Σ is the alphabet of transition labels; δ is the

transition function $Q \to 2^{2^\Sigma \times Q}$; $I \subseteq Q$ is the set of initial states; $T \subseteq Q$ is the set of accepting states.

Note that we use 2^Σ in the definition of the transition relation in place of Σ in order to gather transitions that differ only by their actions — this can be a significant optimisation.

A run of a Büchi automaton is a path through the automaton; it is accepting if the states in T are visited an infinite number of times. That is,

Definition 2.2 A *run* of a Büchi automaton $\mathcal{B}$ with respect to a word $u_0 u_1 \ldots \in \Sigma^\omega$ is a sequence of states in $q_0 q_1 \ldots \in Q^\omega$ with $q_0 \in I$ and $\forall i \exists \alpha_i \, \langle \alpha_i, q_{i+1} \rangle \in \delta(q_i)$ such that $u_i \in \alpha_i$. A run is *accepting* if infinitely many states in the run are members of T.

A *generalised* Büchi automaton (GBA) has a set of accepting sets $\mathcal{T} \subseteq 2^Q$; each set must be visited infinitely often for acceptance. A GBA may be reduced to a classical Büchi automaton but incurs a linear blowup of $O(|\mathcal{T}|)$.

2.4 Alternating Automata

Alternating automata are a type of tree automaton (runs are described as trees rather than linear traces) combining both deterministic and nondeterministic behaviours: a transition in a nondeterministic automaton leads to a set of states from which one is chosen; a transition in a deterministic tree automaton leads to a successor set. Alternating automata exhibit the combination of these existential and universal behaviours. Although the presentation that we adopt below is one of a nondeterministic choice between conjunctions of states, it can be generalised to arbitrary propositional formulæ over $\wedge, \vee$ and states. Alternating automata are exponentially more succinct than Büchi automata.

There are two presentations of LTL to automata conversion via alternating automata. We follow the slightly unconventional presentation by Gastin and Oddoux [11]: transitions are from a state to a conjunction of states; each state may have multiple transitions, selected nondeterministically. This effectively encodes a disjunction of conjunctions of states reached from a given state.

The presentation given by Fritz and Wolper [10] is equivalent, but the differences in the definitions lead to larger representations of the automata. An additional difference is that Gastin and Oddoux use a co-Büchi accepting condition, while Fritz uses a Büchi condition. We can disregard this: for the alternating automata under consideration, a Büchi condition $F \subseteq Q$ is equivalent to the co-Büchi condition $Q \setminus F$.

Definition 2.3 An *alternating co-Büchi automaton* $\mathcal{A}$ is defined by the tuple $\langle Q, \Sigma, \delta, I, F \rangle$ where Q is the set of states; Σ is the alphabet of transition labels; δ is the transition function $Q \to 2^{2^\Sigma \times 2^Q}$; $I \subseteq 2^Q$ is the set of initial combinations of states; $F \subseteq Q$ is the set of final states

As for the Büchi automaton definition above, the transition labels are from 2^Σ; accepted words are nevertheless from Σ^ω.

Alternating automata representing LTL formulæ are known to be *very weak*, which means that there is a partial order on the states $(Q, \sqsubseteq)$ determined by the transitions, such that $\forall q \in Q, \forall \langle \alpha, q' \rangle \in \delta(q), q' \sqsubseteq q$. That is, transitions are only permitted from a state to a lower or equal state. The result of this restriction is that the only loops in very weak co-Büchi alternating automaton (VWAA) are self-loops.

Definition 2.4 A run σ of a VWAA on a word $u_0 u_1 \ldots \in \Sigma^\omega$ is a labelled DAG $\langle V, E, \lambda \rangle$ with V partitioned into levels V_i, $V = \bigcup_{i \in \mathbb{N}} V_i$ and $E \subseteq \bigcup_{i \in \mathbb{N}} V_i \times V_{i+1}$. $\lambda : V \to Q$ labels the vertices of the graph with states of the automaton. V_i may be seen as a multiset of elements of Q. The graph is related to the word and the automaton by $\lambda(V_0) \in I$ and $\forall v \in V_i, \exists \langle \lambda(v), \alpha, s' \rangle \in \delta(\lambda(v)).u_i \in \alpha \wedge s' = \lambda(E(v))$ A run is accepting if every infinite branch of σ has only a finite number of nodes with labels in F.

2.4.1 LTL to VWAA Conversion

We report here the conversion procedure given by Gastin and Oddoux. The set operator $\otimes$ constructs the conjunctions of two sets of disjunctive normal form transitions: $X \otimes Y = \{\langle \alpha_1 \cap \alpha 2, e_1 \wedge e_2 \rangle \mid \langle \alpha_1, e_1 \rangle \in X, \langle \alpha_2, e_2 \rangle \in Y\}$. The overbar operator $\bar{\psi}$ converts ψ to a set-style disjunctive normal form representation: a set of conjunctions of atomic propositions or temporal subformulæ.

For an LTL formula φ over atomic propositions P, the VWAA $\mathcal{A}_\varphi = \langle Q, \Sigma, \delta, I, F \rangle$ is given by

- Q is the set of temporal subformulæ of Q (the set of subformulæ with an LTL operator as the main connective, union the set of atomic propositions)
- $\Sigma = 2^P$; $I = \bar{\psi}$; F is the set of formulæ of the form $\psi_1 \,\mathbf{U}\, \psi_2$ or $\mathbf{F}\, \psi_1$
- δ is defined as

$$
\begin{aligned}
\delta(\top) &= \{\langle \Sigma, \top \rangle\} \\
\delta(p) &= \{\langle \{a \in \Sigma \mid p \in a\}, \top \rangle\} \\
\delta(\neg p) &= \{\langle \{a \in \Sigma \mid p \notin a\}, \top \rangle\} \\
\delta(\mathbf{X}\, \psi) &= \{\langle \Sigma, e \rangle \mid e \in \bar{\psi}\} \\
\delta(\mathbf{F}\, \psi) &= \Delta(\psi) \cup (\{\langle \Sigma, \mathbf{F}\, \psi \rangle\}) \\
\delta(\mathbf{G}\, \psi) &= \Delta(\psi) \otimes \{\langle \Sigma, \mathbf{G}\, \psi \rangle\}) \\
\delta(\psi_1 \,\mathbf{U}\, \psi_2) &= \Delta(\psi_2) \cup (\Delta(\psi_1) \otimes \{\langle \Sigma, \psi_1 \,\mathbf{U}\, \psi_2 \rangle\}) \\
\delta(\psi_1 \,\mathbf{R}\, \psi_2) &= \Delta(\psi_2) \otimes (\Delta(\psi_1) \cup \{\langle \Sigma, \psi_1 \,\mathbf{R}\, \psi_2 \rangle\})
\end{aligned}
$$

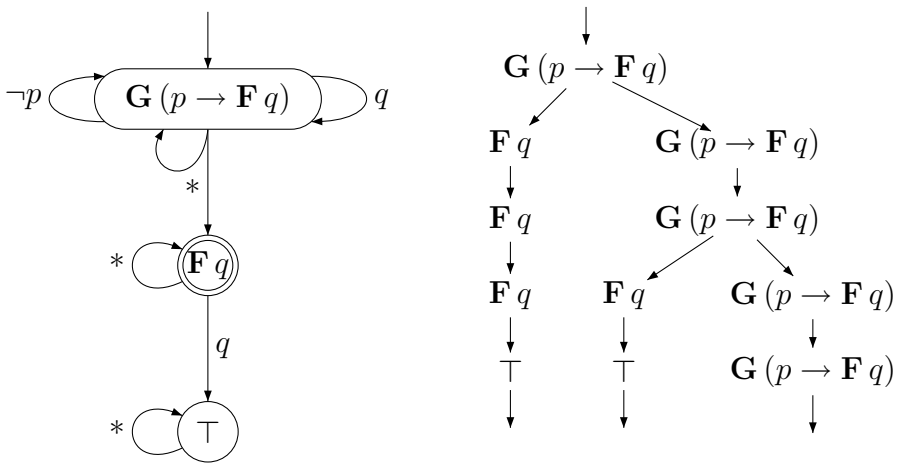


Fig. 2. Example alternating automaton (left) and part of a run over the input $\sigma = \{p\}\{\}\{p\}\{pq\}\cdots$ (right). * indicates the unconstrained transition.

where Δ is the extension of δ to include the propositional subformulæ of φ:

$$\begin{aligned} \Delta(\psi) &= \delta(\psi) && \text{if } \psi \in Q \\ \Delta(\psi_1 \wedge \psi_2) &= \Delta(\psi_1) \otimes \Delta(\psi_2) \\ \Delta(\psi_1 \vee \psi_2) &= \Delta(\psi_1) \cup \Delta(\psi_2) \end{aligned}$$

We give an example VWAA corresponding to the LTL formula $\mathbf{G}\,(p \rightarrow \mathbf{F}\,q)$ in Figure 2 along with a sample run.

2.4.2 Compact Representation of Runs

The representation of a run of a VWAA as a DAG is problematic as the number of vertices at each level grows without bound. We can reduce the representation of a run by restricting each level to a set rather than a multiset, forming a reduced DAG. We call successive sets *configurations*, $C_i \subseteq Q$. A sequence of configurations over a word $u_0 u_1 \ldots \in \Sigma^\omega$ is accepting if there exists a set of edges E partitioned into $E_i \subseteq C_i \times C_{i+1}$ such that $\forall q \in C_i \exists \langle \alpha, q' \rangle \in \delta(q).u_i \in \alpha \wedge q' \subseteq E_i(q)$ and every path $q_0 q_1 \ldots$ such that $q_{i+1} \in E_i(q_i)$ contains only finitely many occurrences of the members of F.

For example, consider the run in Figure 2. The corresponding sequence of configurations is

$$\begin{aligned} &\{\mathbf{G}\,(p \rightarrow \mathbf{F}\,q)\} \\ &\{\mathbf{F}\,q, \mathbf{G}\,(p \rightarrow \mathbf{F}\,q)\} \\ &\{\mathbf{F}\,q, \mathbf{G}\,(p \rightarrow \mathbf{F}\,q)\} \\ &\{\mathbf{F}\,q, \mathbf{G}\,(p \rightarrow \mathbf{F}\,q)\} \\ &\{\mathbf{G}\,(p \rightarrow \mathbf{F}\,q)\} \end{aligned}$$

This is significantly weaker than the original formulation, but we can show that the languages accepted are equivalent. Firstly, every accepting sequence

of configurations C_i with acceptance described by edges E_i may be directly translated into the DAG $\langle \bigcup_{i\in\mathbb{N}} C_i, \bigcup_{i\in\mathbb{N}} E_i, I\rangle$ where I is the identity function on states. In the opposite direction, every accepting DAG can be reduced to an accepting run of configurations given by $C_i = \bigcup_{v\in V_i} \lambda(v)$. We show this sequence is accepting by appealing to an important property of accepting paths: they are both left-append and suffix closed — that is, a suffix of an accepting path is also accepting, as is an accepting path prefixed with a finite number of additional states. This means that the acceptance condition can on configurations can be reduced to the existence of an accepting path from each element of each C_i. This is assured by examination of the DAG, since every element of each V_i must be followed by an accepting sequence of edges.

2.4.3 Superset Property of Runs

Both formulations of runs describe the minimal elements (or multiset) of states at each point in time, but neither requires that the set consists solely of these elements. We may, without changing the language accepted, replace C_i with a superset of C_i (similarly V_i) provided that successive configurations (levels of the tree) can be modified to accommodate the evolution of the extra states while remaining consistent with the definitions of the runs. This is crucial to the encoding described below: we need only constrain the current configuration to be any superset of that described by the transitions.

3 Bounded Model Checking Encodings

Having discussed three representations of LTL formulæ suited to model checking we now turn to the way that these representations can be used for bounded model checking. The encoding of Büchi automata was discussed by de Moura et al. [6] as well as Clarke et al. [5]. The use of SNF for bounded model checking was the subjection of a paper by Frisch et al. [9]. The approach that we take here to bring the encodings together is to isolate the components of the encoding of the specification into three parts: that which constrains the path in all cases; that which constrains the path only when it is a finite path prefix; and that which constrains the path only when it is a k-loop. The addition of the first constraint to the original approach [1] has the potential to simplify the resulting formula[3] considerably:

$$[\![M, f]\!]_k := [\![M]\!]_k \wedge \mathrm{enc}_c(f,k) \wedge \left(\mathrm{enc}_n(f,k) \vee \bigvee_{l=0}^{k} ({}_l\mathrm{L}_k \wedge \mathrm{enc}_l(f,k,l))\right)$$

where $\mathrm{enc}_c, \mathrm{enc}_n$, and enc_l denote the common, finite, and loop encodings as described below.

[3] The formula given is derived from the usual BMC formulation as given in Biere et al. [1]. We write $[\![M]\!]_k$ for the encoding of the model, ${}_l\mathrm{L}_k$ for the constraint that the path is a k-l-loop, but we omit the $\bigwedge_{0\leq l<k} \neg {}_l\mathrm{L}_k$ non-loop constraint as suggested by [3]

3.1 Bounded Model Checking with Büchi Automata

We present a variation on the encoding of de Moura et al. [6], making explicit the representation of states in order to avoid the overhead of enforcing mutual exclusion on states. In contrast with other presentations, we use generalised Büchi automata: the complexity of checking multiple acceptance sets is much lower than the overhead of conversion to classical Büchi automata.

All paths accepted by a Büchi automaton are infinite — formulæ with finite counterexamples such as $\mathbf{F}\,\phi$ are encoded with a trivial infinite loop. The finite prefix case is therefore never accepting, and we deduce that $\mathrm{enc}_n(f,k) = \bot$.

Given a generalised Büchi automaton representing LTL formula f, $\mathcal{B}_f = \langle Q, \Sigma, \delta, I, \mathcal{T}\rangle$, we encode the current state $q \in Q$ as a base two integer in the range $0 \ldots |Q|-1$: there is a one-to-one mapping $\epsilon \subseteq Q \times \{i \mid 0 \leq i < |Q|-1\}$. That is, for each state i, we have a set of propositional variables $q_n(i)$, $0 \leq n < \lceil \log_2(|Q|)\rceil$ and we write $[\![q]\!]^i$ for the assertion that the bit pattern $q_0 q_1 \ldots$ is the base two representation of $\epsilon(q)$. For Büchi automata representing LTL, Σ is the set of propositions in that model; the encoding of elements $a \in \Sigma$ is given as $[\![a]\!]^i$ as for the standard encoding.

The transition relation is encoded as a set of constraints on the originating state, target state, and label. If the transition relation is total, we can write

$$T_{\mathcal{B}_f}(i,k) = \bigvee_{\langle s,\alpha,s'\rangle\in\delta} \bigvee_{a\in\alpha} \left(([\![s]\!]^i \wedge [\![a]\!]^i \wedge [\![s']\!]^{i+1}\right)$$

The initial set is encoded directly as a disjunction over members of I:

$$I_{\mathcal{B}_f}(k) = \bigvee_{s\in I} [\![s]\!]^0$$

Finally, we encode the acceptance sets. The Büchi acceptance condition is that each member of $\mathcal{T}$ is visited infinitely often. As we have ruled out finite path prefixes, we know that all paths being considered are of the form ab^ω. If we assert as part of the loop encoding that the corresponding paths in the Büchi automaton follow the same pattern, we can simply require that representatives from each acceptance set appear in the loop (ie, in b):

$$F_{\mathcal{B}_f}(k,l) = \bigwedge_{T\in\mathcal{T}} \bigvee_{i=l}^{k} \bigvee_{s\in T} [\![s]\!]^i$$

Thus we have

$$\begin{aligned}
\mathrm{enc}_c(f,k) &= I_{\mathcal{B}_f}(k) \wedge \bigwedge_{i=0}^{k-1} T_{\mathcal{B}_f}(i,k) \\
\mathrm{enc}_n(f,k) &= \bot \\
\mathrm{enc}_l(f,k,l) &= F_{\mathcal{B}_f}(k,l) \wedge \bigwedge_{i=0}^{\lceil \log_2(|Q|)\rceil - 1} q_i(l) \leftrightarrow q_i(k)
\end{aligned}$$

Although the LTL to Büchi automaton conversion is exponential in the size of the formula, the encoding above introduces only a linear number of variables. The resulting formula is linear size in the product of the number of transitions and k except for $F_{\mathcal{B}_f}$ which is quadratic: $O(|\mathcal{T}|k^2)$.

3.2 Bounded Model Checking with Alternating Automata

The encoding of alternating automata is very similar to Büchi automata. Since a run is a sequence of configurations rather than states we use one state variable to represent each state; configurations are then represented by conjunctions of states.

Given a VWAA representing LTL formula f, $\mathcal{A}_f = \langle Q, \Sigma, \delta, I, F\rangle$, we encode the presence of a a state q in the ith configuration by the variable $q(i)$. A configuration is encoded as a conjunction of its members: we write $[\![C]\!]^i = \bigwedge_{q\in C} q(i)$, with $[\![\emptyset]\!]^i = \bot$. Note that this constrains the necessary, but not sufficient, members of the configuration, and so describes the smallest configuration that describes the run as discussed in Section 2.4.3. The targets of transitions can be seen as subsets of configurations and are hence encoded in the same way.

For VWAAs derived from LTL formulæ as above, the transitions are labelled with a set of sets of atomic propositions: the set of permitted assignments to propositions. These can be denoted [4] by a conjunction of literals where $p \wedge q$ denotes $\{a \in \Sigma \,|\, p \in a\} \cap \{a \in \Sigma \,|\, q \in a\}$. We write $[\![\alpha]\!]^i$ for the conjunction of literals representing $\alpha \in 2^\Sigma$ — this is particularly convenient as the implementation of the LTL to VWAA conversion [11] produces these conjunctions directly.

As before, the transition relation is given as a series of constraints

$$T_{\mathcal{A}_f}(i,k) = \bigwedge_{q\in Q}\left(q(i) \rightarrow \bigvee_{\langle\alpha,q'\rangle\in\delta(q)} \left([\![\alpha]\!]^i \wedge [\![s']\!]^{i+1}\right)\right)$$

and the initial set of configurations is encoded

$$I_{\mathcal{A}_f}(k) = \bigvee_{C_0\in I} [\![C_0]\!]^0$$

[4] See Remark 2 in Gastin and Oddoux [11]

A VWAA run is accepting if no branch contains an infinite occurrence of elements of F. This can be assured on a k-prefix path if the empty configuration is reached at any point: the *very weak* property means that all successive configurations are also empty and hence no state is visited infinitely often. This also means that we can reduce the check to an empty kth configuration: this will hold even if the first empty configuration is before k.

$$P_{\mathcal{A}_f}(k) = \bigwedge_{q \in Q} \neg q(k)$$

For the loop case, we cannot simply check for an infinite number of occurrences of the members of F as the co-Büchi condition is on paths through the configuration space. That is, an accepting run could consist of an infinite number of paths each with a finite number of occurrences of an acceptance state. In this case the acceptance state would appear in a configuration within the loop suggesting that the state was visited infinitely often. In fact, we must make use of the *very weak* condition again: the only loops in VWAAs are self-loops, and hence the only paths that visit a state infinitely often must do so by always taking the self-loop transition. By the left-append and prefix closed property of accepting paths, we can deduce that if it is possible to take a non-self-loop transition from an accepting state then that state must be part of an accepting path.

$$F_{\mathcal{A}_f}(k,l) = \bigwedge_{q \in F} \bigvee_{i=l}^{k} \left([\![q]\!]^i \rightarrow \bigvee_{\substack{\langle \alpha, q' \rangle \in \delta(q) \\ q \notin q'}} \left([\![\alpha]\!]^i \wedge [\![q']\!]^{i+1} \right) \right)$$

Thus we have

$$\begin{aligned} \mathrm{enc}_c(f,k) &= I_{\mathcal{A}_f}(k) \wedge \bigwedge_{i=0}^{k-1} T_{\mathcal{A}_f}(i,k) \\ \mathrm{enc}_n(f,k) &= P_{\mathcal{A}_f}(k) \\ \mathrm{enc}_l(f,k,l) &= F_{\mathcal{A}_f}(k,l) \wedge \bigwedge_{q \in Q} q(l) \leftrightarrow q(k) \end{aligned}$$

This encoding produces a linear number of variables in the size of the LTL formula. The resulting propositional formula is linear in the product of the number of transitions and k, again except for $F_{\mathcal{A}_f}$ which is quadratic in k.

3.3 Bounded Model Checking with SNF

As SNF is a specialisation of LTL we could encode it using the standard BMC method, but we can produce a much better result by considering the structure of rules. Given a set of rules representing an LTL formula f, Ψ_f, we consider each type of rule separately:

Initial rules $(\mathbf{start} \rightarrow f)$ specify initial conditions:

$$I_{\Psi_f}(k) = \bigwedge_{(\mathbf{start} \rightarrow f) \in \Psi_f} [\![f]\!]^0$$

Global invariant rules $p \rightarrow f$ are constraints on the configurations of individual states:

$$P_{\Psi_f}(i) = \bigwedge_{(p \rightarrow f) \in \Psi_f} [\![p \rightarrow f]\!]^i$$

Global step rules $(p \rightarrow \mathbf{X}\, f)$ connect states with their successors, similar to a transition relation. Above, we included the transition relation in enc_c together with a loop condition on its states in enc_l. However, we can simplify this by isolating the common cases at time $< k$ from the boundary cases which distinguish the behaviour of the finite prefix and k-loop conditions.

$$\begin{aligned} T_{\Psi_f}(i,k) &= \bigwedge_{(p \rightarrow \mathbf{X} f) \in \Psi_f} [\![p]\!]^i_k \rightarrow [\![f]\!]^{i+1} \\ T^n_{\Psi_f}(k) &= \bigwedge_{(p \rightarrow \mathbf{X} f) \in \Psi_f} [\![p]\!]^k_k \rightarrow \bot \\ T^l_{\Psi_f}(k,l) &= \bigwedge_{(p \rightarrow \mathbf{X} f) \in \Psi_f} [\![p]\!]^k_k \rightarrow [\![f]\!]^l \end{aligned}$$

Global eventuality rules $(p \rightarrow \mathbf{F}\, f)$ are superficially similar to acceptance conditions but can be interpreted more directly — as in [1]. For a finite prefix, this is simply a disjunction over states; for a k-loop of the form ab^ω, evaluating $\mathbf{F}$ during b is equivalent to evaluating it at the start of b.

$$\begin{aligned} F^n_{\Psi_f}(k) &= \bigwedge_{i=0}^{k} \bigwedge_{(p \rightarrow \mathbf{F} f) \in \Psi_f} \left([\![p]\!]^i \rightarrow \bigvee_{j=i}^{k} [\![f]\!]^j \right) \\ F^l_{\Psi_f}(k,l) &= \bigwedge_{i=0}^{k} \bigwedge_{(p \rightarrow \mathbf{F} f) \in \Psi_f} \left([\![p]\!]^i_k \rightarrow \bigvee_{j=\min(i,l)}^{k} [\![f]\!]^j \right) \end{aligned}$$

Thus we have

$$\begin{aligned} \mathrm{enc}_c(f,k) &= I_{\Psi_f}(k) \ \wedge \ \bigwedge_{i=0}^{k} P_{\Psi_f}(i) \ \wedge \ \bigwedge_{i=0}^{k-1} T_{\Psi_f}(i,k) \\ \mathrm{enc}_n(f,k) &= T^n_{\Psi_f}(k) \ \wedge \ F^n_{\Psi_f}(k) \\ \mathrm{enc}_l(f,k,l) &= T^l_{\Psi_f}(k,l) \ \wedge \ F^l_{\Psi_f}(k,l) \end{aligned}$$

As noted above, the size of the SNF representation is linear in the size of the LTL formula; the number of variables in the encoding is therefore linear in the product of k and the size of the formula. The size of the resulting formula

using the encoding given above is linear in the product of k and the size of the LTL except for the encoding of eventuality rules which is quadratic in k.

3.3.1 Reduced SNF: the "Fixpoint" form

A further refinement that can be made to SNF in the context of bounded time is the transformation for $\mathbf{F}$ [9]. Using the **bound** operator, which holds only at time k (at the end of the first occurrence of ab in ab^ω), we can write the transformation

$$\textsc{Snf}'_{[\mathbf{F}]}(\{\varphi \rightarrow \mathcal{F}(\mathbf{F}\, f)\} \cup \Gamma) \doteq \left\{ \begin{array}{c} \varphi \rightarrow \psi(f \vee \underline{\mathbf{X}\,\mathbf{F}\ f}) \\ \underline{\mathbf{X}\,\mathbf{F}\ f} \rightarrow \mathbf{X}\,(f \vee \underline{\mathbf{X}\,\mathbf{F}\ f}) \\ \mathbf{bound} \rightarrow \neg\underline{\mathbf{X}\,\mathbf{F}\ f} \end{array} \right\} \cup \Gamma$$

This direct approach affects the length of the counterexample: evaluating $\mathbf{F}\, x$ in the loop part of the path will only check states up to k, rather than the whole of the loop as expected. The direct encoding approach to this is to always evaluate an eventuality from the start of the loop, since $\bigvee_{i=n}^{k} x \vee \bigvee_{i=l}^{n-1} x \equiv \bigvee_{i=l}^{k} x$ for $l < n \leq k$. The equivalent for RSNF is to consider each eventuality at both the current time and projected to the start of the loop. The latter is explicitly renamed out and the projection asserted by the AtLoop rule given below. This renaming is *time-independent*; that is, the introduced variable $\underline{\mathbf{F}\ f}$ is not a state variable but rather is a simple propositional variable, and this is reflected in the encoding.

$$\textsc{Snf}''_{[\mathbf{F}]}(\{\varphi \rightarrow \mathcal{F}(\mathbf{F}\, f)\} \cup \Gamma) \doteq \left\{ \begin{array}{c} \varphi \rightarrow \psi(f \vee \underline{\mathbf{X}\,\mathbf{F}\ f} \vee \underline{\mathbf{F}\ f}) \\ \underline{\mathbf{F}\ f} \rightarrow \textsc{AtLoop}(f \vee \underline{\mathbf{X}\,\mathbf{F}\ f}) \\ \underline{\mathbf{X}\,\mathbf{F}\ f} \rightarrow \mathbf{X}\,(f \vee \underline{\mathbf{X}\,\mathbf{F}\ f}) \\ \mathbf{bound} \rightarrow \neg\underline{\mathbf{X}\,\mathbf{F}\ f} \end{array} \right\} \cup \Gamma$$

This gives us the encoding

$$\begin{aligned} \textsc{enc}'_c(f,k) &= \textsc{enc}_c(f,k) \ \wedge \bigwedge_{(\mathbf{bound} \rightarrow f) \in \Psi_f} [\![f]\!]^k \\ \textsc{enc}'_n(f,k) &= T^n_{\Psi_f}(k) \ \wedge \bigwedge_{(p \rightarrow \textsc{AtLoop}(f)) \in \Psi_f} p \rightarrow \bot \\ \textsc{enc}'_l(f,k,l) &= T^l_{\Psi_f}(k,l) \ \wedge \bigwedge_{(p \rightarrow \textsc{AtLoop}(f)) \in \Psi_f} p \rightarrow [\![f]\!]^l \end{aligned}$$

For an alternative presentation of this approach, see Cimatti et al. [4].

The encoding given above has the number of variables linear in the product of k and the size of the formula as before. The size of the resulting formula is linear in the product of k and the size of the LTL.

4 SNF versus Automata

We have examined two established methods of encoding LTL for bounded model checking and introduced a third: the encoding via alternating automata. We now clarify the relationships and relative advantages of the encodings.

4.1 SNF and Alternating Automata

The configuration view of alternating automata makes it apparent that Fixpoint and AA are nearly equivalent. Step rules in SNF/Fixpoint relate states and their successors to the evolved state of the model, while AA transitions which relates states and their successors to the present state of the model. We can project each SNF variable x created during LTL conversion to a VWAA state $\mathbf{X}\,x$: the set of SNF variables is directly related to the members of the configurations of the VWAA. Furthermore, we can show that SNF step rules created from LTL always have atomic antecedents: a necessary condition to relate step rules to transitions.

The boundary condition used in Fixpoint to represent eventualities corresponds to an assertion that x occurs finitely, not infinitely, often. It is introduced for the same states that, in the alternating automaton conversion, would be in the co-Büchi acceptance set. The difficulty of checking the co-Büchi acceptance condition are sidestepped by the start-of-loop projection introduced in Section 3.3.1. Effectively, all branches of the run are collapsed into one.

In fact, this is the main advantage of SNF over VWAAs: the encoding of the acceptance set is complex and comparatively large for the alternating automaton encoding. There are other advantages: not being a transition system, the variables introduced by SNF are not included in the loopback condition L_k, eliminating the need for the empty-configuration assertion in the finite case. This can even reduce slightly the bound at which counterexamples are found. Alternating automata do benefit from the simplification [11] and simulation [10] reductions, some of which do not project directly to SNF; the advantages of these have the potential to outweigh the drawbacks of the encoding.

4.2 SNF and Alternating Automata versus Büchi Automata

Most of the encoding issues discussed above apply equally to Büchi automata, the exception being the acceptance set which is simpler than the alternating case, although still more complex than the Fixpoint case. The biggest drawback for BMC is the requirement for an infinite path. Safety properties with finite counterexamples must still end up in a loop — in both the specification automaton *and the model* which could lengthen the counterexample considerably. In fact, the best choice for simple specifications seems to be the direct

encoding: in such a case, the loop constraint could be eliminated altogether.

There are two other loop-related problems with the use of Büchi automata. Firstly, when both the specification and model automata must be in a loop, the length of the loop is the least common multiple of the lengths of the loops in the two automata on their own. This is not an issue for alternating automata because of the weakness property: all loops will be a single state. Secondly, BMC is able to take special advantage of the loopback where a finite counterexample takes the form ab^i. For example, consider the word $xx(abb)^\omega$, which is recognised in this form by the specification $F(b \wedge F(a))$ using the direct or SNF encodings, but which must be expanded to $xxabb(abb)^\omega$ to be recognised by the automata-based encodings.

4.3 Complexity

We noted the complexity of each encoding at the end of its corresponding section. In each case, the encoding produces a linear number of variables and symbols in the size of the original LTL, and in each case there is only a small part of the encoding which produces a quadratic, rather than linear, number of symbols in k: for the automata encodings, it is the Büchi and co-Büchi conditions; for SNF it is the encoding of eventualities. The refinement of SNF can, however, be encoded in a linear number of symbols as described above. No such improvement is immediately obvious for the automata encodings, so specifications including **R** or **G** operators suffer from quadratic growth with these encodings.

4.4 Empirical Results

To demonstrate some of the differences between the approaches we give a selection of experimental results comparing a variety of BMC encodings. The existing encodings, the original BMC encoding [1] (marked "Orig" in the results), the SNF encoding and its refinement [9] ("SNF" and "FIX") are compared against Büchi automata, in this case the Etessami and Holzmann [7] procedure ("TMP"), and the VWAA produced by the tool from Gastin and Oddoux [11] with and without its simplifications ("AA" and "AA-").

To provide a comparison over a range of LTL specifications we fix the model for the experiments, using a distributed mutual exclusion example [14] with the specifications given in Frisch et al. [9], at several different bounds to illustrate scalability. The number and nesting depths of temporal operators appearing in the specifications are reported as pairs of numbers alongside their names in the tables. We used a modified version of NuSMV [2] with an improved CNF conversion [16]; timings were made in the SAT solver zChaff [15].

Table 1 shows the results from verifying three correct specifications. Rather than report the number of states that each automata conversion produces, we report the size of the CNF result. This means that the automaton methods can be directly compared to the SNF and direct encodings.

Enc.	k	Size	Vars	Time	Size	Vars	Time	Size	Vars	Time
		Accessibility (4,2)			Overtaking 1 (5,5)			Overtaking 2 (8,8)		
	30	14596	2480	0.35	15737	2511	0.17	16339	2573	0.40
AA	40	19436	3280	1.47	20957	3321	2.02	21759	3403	1.05
	50	24276	4080	4.67	26177	4131	8.45	27179	4233	10.11
	30	15325	2759	0.39	17011	2945	0.35	18065	3069	0.35
AA-	40	20405	3649	1.39	22651	3895	1.40	24055	4059	1.35
	50	25485	4539	5.28	28291	4845	11.87	30045	5049	7.23
	30	14298	2418	0.97	14481	2480	0.23	14814	2573	0.25
SNF	40	19038	3198	0.90	19281	3280	0.75	19724	3403	1.08
	50	23778	3978	4.04	24081	4080	2.76	24634	4233	2.22
	30	14299	2449	0.72	14483	2511	0.21	14816	2604	0.27
FIX	40	19039	3239	0.89	19283	3321	0.96	19726	3444	0.88
	50	23779	4029	4.43	24083	4131	4.17	24636	4284	2.59
	30	14599	2418	0.75	16559	2449	0.42	17898	2480	0.46
TMP	40	19439	3198	4.54	22049	3239	1.43	23828	3280	1.24
	50	24279	3978	4.90	27539	4029	3.54	29758	4080	7.07
	30	15848	2356	0.26	41874	2356	0.37	Encoding time		
Orig	40	21908	3116	1.47	81539	3116	1.92	$>$ 1800 secs		
	50	28368	3876	9.83	142404	3876	17.69			

Table 1
Timings in zChaff for the DME example using three valid specifications. Specifications given as "Name (number of temporal operators, maximum nesting depth)"; "Size" indicates the number of clauses.

We observe that as the specifications become more complex, the simplicity of the SNF encoding has an increasing advantage. The alternating automata approach lags close behind the Büchi automata produced by TMP: a particularly interesting result, as the latter includes advanced simulation-based simplification techniques, while the former uses simple transition and state simplifications.

We illustrate the effect of the different encodings on counterexample size by comparing two incorrect specifications with different minimal counterexamples (Table 2). Here we see that the Büchi automaton procedure is slower due to the longer counterexample produced. The other procedures are all comparable although the VWAA method is slightly faster on the larger example.

Enc.	k	Time	k	Time
	Priority 1 (4,2)		Priority 2 (4,2)	
AA	14	0.03	53	0.30
AA-	14	0.03	53	0.89
SNF	13	0.02	52	0.49
FIX	13	0.02	52	0.83
TMP	53	3.26	> 200	
Orig	13	0.02	52	1.15

Table 2
Timings in zChaff for the DME example using two invalid specifications. Specifications given as "Name (number of temporal operators, maximum nesting depth)"

5 Conclusions and Future Work

The main advantage of automata based bounded model checking, the high state of development of the conversion procedures, is balanced by the numerous drawbacks of conversion. We have described how the use o alternating automata overcomes many of these problems and demonstrated their use for BMC. A simple alternating automata encoding has been shown to be almost as effective as a highly developed Büchi automata approach, although both lag behind the SNF encoding (without any simplification) on many of the examples given.

This work has indicated several promising directions for further development. Simulation-based simplification for alternating automata [10] may improve the performance of the approach, and the close relationship with SNF could mean that the SNF encoding could also be improved by such simplification techniques. This relationship could also yield better encodings for the co-Büchi condition, further improving the performance. A possible alternative technique for encoding the co-Büchi condition is to adapt the new linear-space encoding of Latvala et al. [13].

6 Acknowledgements

I would not have begun to investigate these encoding methods without the helpful comments from the reviewers for CHARME 2003. I am also indebted to Dr Paul Jackson for extensive discussion and input on the topics in this paper.

References

[1] Biere, A., A. Cimatti, E. Clarke and Y. Zhu, *Symbolic model checking without BDDs*, in: W. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems. 5th International Conference, TACAS'99*, Lecture Notes in Computer Science **1579** (1999), pp. 193–207.

[2] Cimatti, A., E. Clarke, F. Giunchiglia and M. Roveri, NuSMV*: a new Symbolic Model Verifier*, in: N. Halbwachs and D. Peled, editors, *Proceedings of the Eleventh Conference on Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science (1999), pp. 495–499.

[3] Cimatti, A., M. Pistore, M. Roveri and R. Sebastiani, *Improving the encoding of LTL model checking into SAT*, in: A. Cortesi, editor, *Third International Workshop on Verification, Model Checking and Abstract Interpretation*, Lecture Notes in Computer Science **2294** (2002), pp. 196–207.

[4] Cimatti, A., M. Roveri and D. Sheridan, *Bounded verification of past LTL*, in: A. J. Hu and A. K. Martin, editors, *Formal Methods in Computer-Aided Design; 5th International Conference, FMCAD 2004*, Lecture Notes in Computer Science (2004).

[5] Clarke, E. M., D. Kroening, J. Ouaknine and O. Strichman, *Completeness and complexity of bounded model checking*, in: B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings*, Lecture Notes in Computer Science **2937** (2004), pp. 85–96.

[6] de Moura, L., H. Rueß and M. Sorea, *Lazy theorem proving for bounded model checking over infinite domains*, in: A. Voronkov, editor, *Automated Deduction - CADE-18; 18th International Conference on Automated Deduction*, Lecture Notes in Computer Science **2392** (2002), pp. 438–455.

[7] Etessami, K. and G. J. Holzmann, *Optimizing Büchi automata*, in: C. Palamidessi, editor, *CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 22-25, 2000, Proceedings*, Lecture Notes in Computer Science **1877** (2000), pp. 153–167.

[8] Fisher, M., *A resolution method for temporal logic*, in: *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI)* (1991), pp. 99–104.

[9] Frisch, A., D. Sheridan and T. Walsh, *A fixpoint based encoding for bounded model checking*, in: M. D. Aagaard and J. W. O'Leary, editors, *Formal Methods in Computer-Aided Design; 4th International Conference, FMCAD 2002*, Lecture Notes in Computer Science **2517** (2002), pp. 238–254.

[10] Fritz, C., *Constructing Büchi automata from linear temporal logic using simulation relations for alternating Büchi automata*, in: O. H. Ibarra and Z. Dang, editors, *Implementation and Application of Automata. Eighth*

International Conference (CIAA 2003), Lecture Notes in Computer Science **2759**, Santa Barbara, CA, USA, 2003, pp. 35–48.

[11] Gastin, P. and D. Oddoux, *Fast LTL to Büchi automata translation*, in: G. Berry, H. Comon and A. Finkel, editors, *Proceedings of the 13th Conference on Computer Aided Verification (CAV'01)*, number 2102 in Lecture Notes in Computer Science (2001), pp. 53–65.

[12] Gerth, R., D. Peled, M. Vardi and P. Wolper, *Simple on-the-fly automatic verification of linear temporal logic*, in: P. Dembinski and M. Sredniawa, editors, *Protocol Specification, Testing and Verification XV, Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification*, IFIP Conference Proceedings **38** (1995), pp. 3–18.

[13] Latvala, T., A. Biere, K. Heljanko and T. Junttila, *Simple bounded LTL model checking*, in: A. J. Hu and A. K. Martin, editors, *Formal Methods in Computer-Aided Design; 5th International Conference, FMCAD 2004*, Lecture Notes in Computer Science (2004).

[14] Martin, A. J., *The design of a self-timed circuit for distributed mutual exclusion*, in: H. Fuchs, editor, *Proceedings of the 1985 Chapel Hill Conference on VLSI* (1985), pp. 245–260.

[15] Moskewicz, M., C. Madigan, Y. Zhao, L. Zhang and S. Malik, *Chaff: Engineering an efficient SAT solver*, in: *39th Design Automation Conference*, Las Vegas, 2001, pp. 530–535.

[16] Sheridan, D., *The optimality of a fast CNF conversion and its use with SAT*, Technical Report APES-82-2002, APES Research Group (2004), available from http://www.dcs.st-and.ac.uk/~apes/apesreports.html.

[17] Wolper, P., *Constructing automata from temporal logic formulas: A tutorial*, in: *Lectures on Formal Methods in Performance Analysis (First EEF/Euro Summer School on Trends in Computer Science)*, Lecture Notes in Computer Science **2090** (2001), pp. 261–277.

Bibliography

[1] Parosh Aziz Abdulla, Per Bjesse, and Niklas Eén. Symbolic reachability analysis based on SAT-solvers. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 6th International Conference, TACAS'00*, volume 1785 of *Lecture Notes in Computer Science*, pages 411–425. Springer-Verlag, March 2000.

[2] B. Aspvall, M. Plass, and R. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8:121–123, 1979.

[3] Audemard, Cimatti, Kornilowicz, and Sebastiani. Bounded model checking for timed systems. In *IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE), LNCS*, volume 22, 2002.

[4] Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Korniłowicz, and Roberto Sebastiani. A SAT based approach for solving formulas over boolean and linear mathematical propositions. In A Voronkov, editor, *Automated Deduction — CADE-18; 18th International Conference on Automated Deduction*, volume 2392 of *Lecture Notes in Computer Science*, pages 195–210, Copenhagen, Denmark, July 2002. Springer-Verlag.

[5] Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of Boolean cardinality constraints. In *Principles and Practice of Constraint Programming — 9th International Conference, CP 2003*, Lecture Notes in Computer Science, 2003.

[6] Howard Barringer, Michael Fisher, Dov Gabbay, Graham Gough, and Richard Owens. MetateM: A framework for programming in temporal logic. In *Proceedings of the Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 94–129. Springer-Verlag, June 1989.

[7] Marco Benedetti and Sara Bernardini. Incremental compilation-to-SAT procedures. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT04)*, volume 3542 of *Lecture Notes in Computer Science*, 2004.

[8] Marco Benedetti and Alessandro Cimatti. Bounded model checking for past LTL. In *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS'03*, Lecture Notes in Computer Science, Warsaw, Poland, April 2003. Springer-Verlag.

[9] Daniel Le Berre and Laurent Simon. Sat competition 2004. `http://www.satlive.org/SATCompetition/2004/`, May 2004.

[10] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Design Automation Conference (DAC'99)*, Proceedings of the 36th ACM/IEEE conference on Design automation, pages 317–320, 1999.

[11] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58, 2003.

[12] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In W.R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems. 5th International Conference, TACAS'99*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, July 1999.

[13] Armin Biere, Edmund Clarke, Richard Raimi, and Yunshan Zhu. Verifying safety properties of a PowerPC[tm] microprocessor using symbolic model checking without BDDs. In N. Halbwachs and D. Peled, editors, *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 60–71. Springer-Verlag, 1999.

[14] Alexander Bolotov. *Clausal Resolution for Branching-Time Temporal Logic*. PhD thesis, Manchester Metropolitan University, July 2000.

[15] Alexander Bolotov and Michael Fisher. A resolution method for CTL branching-time temporal logic. In *Proceedings of the Fourth International Workshop on Temporal Representation and Reasoning (TIME)*. IEEE Press, 1997.

[16] Alexander Bolotov, Michael Fisher, and Clare Dixon. On the relationship between ω-qutomata and temporal logic normal forms. *Journal of Logic and Computation*, 2001.

[17] Thierry Boy de la Tour. Minimizing the number of clauses by renaming. In Mark E. Stickel, editor, *10th International Conference on Automated Deduction*, volume 449 of *Lecture Notes in Artificial Intelligence*, pages 558–572. Springer-Verlag, 1990.

[18] Thierry Boy de la Tour. An optimality result for clause form translation. *Journal of Symbolic Computation*, 14:283–301, 1992.

[19] Ron Breukelaar, Erik D. Demaine, Susan Hohenberger, Hendrik Jan Hoogeboom, Walter A. Kosters, and David Liben-Nowell. Tetris is hard, even to approximate. *International Journal of Computational Geometry & Applications*, 14(1–2):41–68, 2004.

[20] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[21] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} states and beyond. *Information and Computing*, 98(2):142–170, 1992.

[22] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Proceedings of the Eleventh Conference on Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 495–499, Trento, Italy, July 1999. Springer-Verlag.

[23] Alessandro Cimatti, Marco Pistore, Marco Roveri, and Roberto Sebastiani. Improving the encoding of LTL model checking into SAT. In Agostino Cortesi, editor, *Third International Workshop on Verification, Model Checking and Abstract Interpretation*, volume 2294 of *Lecture Notes in Computer Science*, pages 196–207. Springer-Verlag, January 2002.

[24] Alessandro Cimatti, Marco Roveri, and Daniel Sheridan. Bounded verification of past LTL. In A J Hu and A K Martin, editors, *Formal Methods in Computer-Aided Design; 5th International Conference, FMCAD 2004*, Lecture Notes in Computer Science, Austin, TX, USA, November 2004. Springer-Verlag.

[25] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[26] E. Clarke, O. Grumberg, and K. Hamaguchi. Another Look at LTL Model Checking. *Formal Methods in System Design*, 10:47–71, 1997.

[27] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[28] Edmund M. Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Completeness and complexity of bounded model checking. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11–13, 2004, Proceedings*, volume 2937 of *Lecture Notes in Computer Science*, pages 85–96. Springer-Verlag, 2004. ISBN 3-540-20803-8.

[29] Stephen Cook. The P vs NP problem. `http://www.claymath.org/millennium/P_vs_NP/`.

[30] Tim B. Cooper and Jeffrey H. Kingston. The complexity of timetable construction problems. In *The Practice and Theory of Automated Timetabling (selected papers from Proceedings of the First International Conference, Practice and Theory of Automated Timetabling, Edinburgh, 1995)*, volume 1153 of *Lecture Notes in Computer Science*, pages 283–295. Springer-Verlag, 1996.

[31] Fady Copty, Limor Fix, Ranan Fraer, Enrico Giunchiglia, Gila Kamhi, Armando Tacchella, and Moshe Y Vardi. Benefits of bounded model checking at an industrial setting. In G Berry, H Comon, and A Finkel, editors, *13th Conference on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 436–453. Springer-Verlag, July 2001.

[32] Olivier Coudert, Jean Christophe Madre, and Christian Berthet. Verifying temporal properties of sequential machines without building their state diagrams. In *CAV '90: Proceedings of the 2nd International Workshop on Computer Aided Verification*, pages 23–32. Springer-Verlag, 1991. ISBN 3-540-54477-1.

[33] Marco Daniele, Fausto Giunchiglia, and Moshe Y. Vardi. Improved automata generation for linear temporal logic. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6–10, 1999, Proceedings*, volume 1633 of *Lecture Notes in Computer Science*, pages 249–260. Springer-Verlag, 1999.

[34] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[35] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[36] Leonardo de Moura, Harald Rueß, and Maria Sorea. Lazy theorem proving for bounded model checking over infinite domains. In A Voronkov, editor, *Automated Deduction — CADE-18; 18th International Conference on Automated Deduction*, volume 2392 of *Lecture Notes in Computer Science*, pages 438–455, Copenhagen, Denmark, July 2002. Springer-Verlag.

[37] Anatoli Degtyarev, Michael Fisher, and Boris Konev. A simplified clausal resolution procedure for propositional linear-time temporal logic. In *Proceedings of Automated Reasoning with Analytic Tableaux and Related Methods*, volume 2381 of *Lecture Notes in Computer Science*, pages 85–99, Copenhagen, Denmark, July 2002.

[38] DIMACS. Suggested satisfiabililty format. Available from `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/`, 1993.

[39] Clare Dixon, Alexander Bolotov, and Michael Fisher. Alternating automata and temporal logic normal forms. *Annals of Pure and Applied Logic*, 135:263–285, 2005.

[40] Clare L. Dixon. *Strategies for Temporal Resolution*. PhD thesis, University of Manchester, September 1995.

[41] Paul E. Dunne. An annotated list of selected NP-complete problems. `http://www.csc.liv.ac.uk/~ped/teachadmin/COMP202/annotated_np.html`.

[42] M.B. Dwyer, G.S. Avruning, and J.C. Corbett. Patterns in property specifications for finite-state verification. In *21st International Conference on Software Engineering, Los Angeles, California*, May 1999.

[43] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In Jan van Leeuwen J. W. de Bakker, editor, *Automata, Languages and Programming, 7th Colloquium*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer-Verlag, 1980. ISBN 3-540-10003-2.

[44] E. Allen Emerson and Joseph Y. Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.

[45] Kousha Etessami and Gerard J. Holzmann. Optimizing Büchi automata. In Catuscia Palamidessi, editor, *CONCUR 2000 — Concurrency Theory, 11th International Conference, University Park, PA, USA, August 22–25, 2000, Proceedings*, volume 1877 of *Lecture Notes in Computer Science*, pages 153–167. Springer-Verlag, 2000.

[46] Michael Fisher. A resolution method for temporal logic. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 99–104. Morgan Kaufmann, August 1991.

[47] Michael Fisher. A normal form for first-order temporal formulae. In D. Kapur, editor, *Eleventh International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 370–384. Springer-Verlag, 1992.

[48] Alan Frisch, Daniel Sheridan, and Toby Walsh. A fixpoint based encoding for bounded model checking. In M D Aagaard and J W O'Leary, editors, *Formal Methods in Computer-Aided Design; 4th International Conference, FMCAD 2002*, volume 2517 of *Lecture Notes in Computer Science*, pages 238–254, Portland, OR, USA, November 2002. Springer-Verlag.

[49] Carsten Fritz. Constructing Büchi automata from linear temporal logic using simulation relations for alternating Büchi automata. In Oscar H. Ibarra and Zhe Dang, editors, *Implementation and Application of Automata. Eighth International Conference (CIAA 2003)*, volume 2759 of *Lecture Notes in Computer Science*, pages 35–48, Santa Barbara, CA, USA, 2003.

[50] Dov Gabbay. The declarative past and imperative future. In H. Barringer, editor, *Proccedings of the Colloquium on Temporal Logic and Specifications*, volume 398 of *Lecture Notes in Computer Science*, pages 409–448. Springer-Verlag, 1989.

[51] M.C. Fernández Gago, M. Fisher, and C. Dixon. Algorithms for guiding clausal temporal resolution. In *Proceedings of KI-2002*, volume 2479 of *Lecture Notes in Artificial Intelligence*, 2002.

[52] M.R. Garey and D.S. Johnson. *Computers and Intractability — A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, San Francisco, 1979.

[53] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th Conference on Computer Aided Verification (CAV'01)*, number 2102 in Lecture Notes in Computer Science, pages 53–65. Springer Verlag, 2001.

[54] Ian P. Gent and Toby Walsh. The SAT phase transition. In *Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI'94)*, pages 105–109, 1994.

[55] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, September 2003.

[56] Rob Gerth, Doron Peled, Moshe Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In Piotr Dembinski and Marek Sredniawa, editors, *Protocol Specification, Testing and Verification XV, Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification*, volume 38 of *IFIP Conference Proceedings*, pages 3–18. Chapman & Hall, June 1995.

[57] Dimitra Giannakopoulou and Flavio Lerda. From states to transitions: Improving translation of LTL formulae to Büchi automata. In Doron .A. Peled and Moshe Y. Vardi, editors, *Proceedings of the 22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2002)*, number 2529 in Lecture Notes in Computer Science, pages 308 – 326, Houston, Texas, November 2002. Springer-Verlag.

[58] Enrico Giunchiglia and Roberto Sebastiani. Applying the Davis-Putnam procedure to non-clausal formulas. In *AI*IA 99: Advances in Artificial Intelligence*, volume 1792 of *Lecture Notes in Artificial Intelligence*, pages 84–94. Springer-Verlag, 2000. ISBN 3-540-67350-4.

[59] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Design, Automation, and Test in Europe (DATE '02)*, pages 142–149, March 2002.

[60] The VIS Group. VIS: A system for verification and synthesis. In R. Alur and T. Henzinger, editors, *Proceedings of the 8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432, New Brunswick, NJ, July 1996. Springer-Verlag.

[61] N. Gupta and D. S. Nau. On the complexity of blocks-world planning. *Artificial Intelligence*, 56:223–254, 1992.

[62] K. Heljanko, T. Junttila, and T. Latvala. Incremental and complete bounded model checking for full PLTL. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'2005)*, volume 3576 of *Lecture Notes in Computer Science*, pages 98–111, 2005.

[63] Paul Jackson and Daniel Sheridan. Clause form conversions for boolean circuits. In *Proceedings of the Seventh International Conference on the Theory and Applications of Satisfiability Testing (SAT 04)*, Lecture Notes in Computer Science, 2004.

[64] Paul B. Jackson. Simplification of QLTL expressions. Draft technical note (personal communication), August 2004.

[65] T. A. Juntilla. Boolean circuit tools (including BCZChaff). `http://www.tcs.hut.fi/~tjuntil/circuits`, May 2003.

[66] Tommi A. Junttila and Ilkka Niemelä. Towards an efficient tableau method for Boolean Circuit Satisfiability Checking. In *Computational logic-CL 2000: First International Conference, London, UK, July 24–28, 2000: proceedings*, volume 1861 of *Lecture Notes in Computer Science and Lecture Notes in Artificial Intelligence*, pages 553–567. Springer-Verlag, 2000. ISBN 3-540-67797-6 (softcover).

[67] Henry Kautz and Bart Selman. Planning as satisfiability. In J. LLoyd, editor, *Proceedings of the Tenth European Conference on Artificial Intelligence*, pages 359–379, 1992.

[68] Henry Kautz and Bart Selman. Unifying SAT-based and graph-based planning. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, volume 1, pages 318–325, 1999.

[69] Richard Kaye. Minesweeper is NP-complete. *Mathematical Intelligencer*, 22(2):9–15, Spring 2000.

[70] T. Latvala, A. Biere, K. Heljanko, and T. Junttila. Simple is better: Efficient bounded model checking for past LTL. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'2005)*, volume 3385 of *Lecture Notes in Computer Science*, pages 380–395, Paris, France, January 2005.

[71] Timo Latvala, Armin Biere, Keijo Heljanko, and Tommi Junttila. Simple bounded LTL model checking. In A J Hu and A K Martin, editors, *Formal Methods in Computer-Aided Design; 5th International Conference, FMCAD 2004*, Lecture Notes in Computer Science, Austin, TX, USA, November 2004. Springer-Verlag.

[72] Monika Maidl. The common fragment of CTL and LTL. In *Proceedings of the 41th Annual Symposium on Foundations of Computer Science*, pages 643–652, 2000.

[73] João P. Marques-Silva and Karem A. Sakallah. Grasp – a new search algorithm for satisfiability. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, November 1996.

[74] A. J. Martin. The design of a self-timed circuit for distributed mutual exclusion. In Henry Fuchs, editor, *Proceedings of the 1985 Chapel Hill Conference on VLSI*, pages 245–260. Computer Science Press, 1985.

[75] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, Edinburgh, 1969.

[76] K. L. McMillan. Interpolation and SAT-based model checking. In Warren A. Hunt, Jr and Fabio Somenzi, editors, *Computer Aided Verification; 15th International Conference, CAV 2003*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Spinger-Verlag, July 2003.

[77] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, 1992.

[78] Ken L McMillan. Applying SAT methods in unbounded symbolic model checking. In E Brinksma and K Guldstrand Larsen, editors, *Computer Aided Verification; 14th International Conference, CAV 2002*, volume 2404, pages 250–264, Copenhagen, Denmark, July 2002.

[79] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *39th Design Automation Conference*, pages 530–535, Las Vegas, June 2001.

[80] Neil V. Murray. Completely non-clausal theorem proving. *Artificial Intelligence*, 18(1):67–85, 1982.

[81] Andreas Nonnengart, Georg Rock, and Christoph Weidenbach. On generating small clause normal forms. In Claude Kirchner and Hélène Kirchner, editors, *Fifteenth International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 397–411. Springer-Verlag, 1998.

[82] Wojciech Penczek, Bożna Wo'zna, and Andrzej Zbrzezny. Towards bounded model checking for the universal fragment of TCTL. In *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, volume 2469 of *Lecture Notes in Computer Science*, 2002.

[83] Wojciech Penczek, Bożna Wo'zna, and Andrzej Zbrzezny. Bounded model checking for the universal fragment of CTL. *Fundamenta Informaticae*, 51(1), May 2003.

[84] David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, September 1986.

[85] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[86] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

[87] Klaus Schneider. Improving automata generation for linear temporal logic by considering the automaton hierarchy. In *Proceedings 8th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, volume 2250 of *Lecture Notes in Computer Science*, pages 39–54, Havana (Cuba), 2001. Springer-Verlag.

[88] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Proceedings of Formal Methods in Computer-Aided Design 2000*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer-Verlag, 2000. ISBN 3-540-41219-0.

[89] Daniel Sheridan. Bounded model checking with SNF, alternating automata and Büchi automata. In *Second International Workshop on Bounded Model Checking*, Electronic Notes in Theoretical Computer Science. Elsevier, July 2004.

[90] Daniel Sheridan. The optimality of a fast CNF conversion and its use with SAT. Technical Report APES-82-2002, APES Research Group, March 2004. Available from `http://www.dcs.st-and.ac.uk/~apes/apesreports.html`.

[91] Daniel Sheridan and Toby Walsh. Clause forms generated by bounded model checking. In Andrei Voronkov, editor, *Eighth Workshop on Automated Reasoning*, 2001.

[92] Ofer Shtrichman. Tuning SAT checkers for bounded model checking. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[93] Fabio Somenzi and Roderick Bloem. Efficient Büchi automata from LTL formulæ. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 248–263. Springer-Verlag, July 2000.

[94] Maria Sorea. Bounded model checking for timed automata. In Walter Vogler and Kim Larsen, editors, *Electronic Notes in Theoretical Computer Science*, volume 68. Elsevier, 2003.

[95] O. Strichman. Pruning techniques for the SAT-based bounded model checking problem. In *Proceeding of the 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'01)*, volume 2144 of *Lecture Notes in Computer Science*, pages 58–70, September 2001.

[96] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[97] H. Tauriainen and K. Heljanko. Testing LTL formula translation into Büchi automata. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(1):57–70, 2002.

[98] G. S. Tseitin. On the complexity of derivation in the propositional calculus. *Zapiski nauchnykh seminarov LOMI*, 8:234–259, 1968. English translation: Consultants Bureau, N.Y., 1970, pp. 115–125.

[99] Moshe Y. Vardi. Branching vs. linear time: Final showdown. In Tiziana Margaria and Wang Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *Lecture Notes in Computer Science*, pages 1–22. Springer-Verlag, April 2001.

[100] Christoph Weidenbach. SPASS: An automated theorem prover for first-order logic with equality. `http://spass.mpi-sb.mpg.de/`, March 2003.

[101] Bożna Woźna and Andrzej Zbrzezny. Reaching the limits for bounded model checking. ICS PAS 958, Warsaw, May 2003.

[102] Hantao Zhang. SATO: An efficient propositional prover. In Mark E. Stickel, editor, *14th International Conference on Automated Deduction*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 272–275. Springer-Verlag, 1997.

[103] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *International Conference on Computer-Aided Design (ICCAD'01)*, pages 279–285, November 2001.

Index

$\Rightarrow$ . . . . . xvi, 73
$\bot$ 16
$[\![]\!]$ 60
$\models$ 39
$\models^0$ 39
$\models^i$ **39**
$\models^i_k$ 51
$\models_F$ 47
$\times$ **131**
$\top$ 16
$\models$ 16
$\langle\!\langle\rangle\!\rangle$ 77

accepting run 36
automaton 35, 177
  Büchi . . . *see* Büchi automaton

Büchi automaton 35, 68
  generalised . . . *see* generalised Büchi automaton
BDD 155
BDLT 139
BDLT′ 148
Boolean circuits 27, 30
Boy de la Tour 27, 129

clause . . . . . *see* CNF, clause
clause 21
clause form . . . . . *see* CNF
CNF 20, **21**
  clause **21**
  conversion **22**, 26
    definitional 26, 33, 132
    for RBCs 33, 129
    standard 132
    structure preserving . 28, 132
  literal **21**
  satisfiability of 21
Cnf 24, 33
cnf 21
Comp 141
complete semantics 53
computational tree logic . *see* CTL, 73
conjunctive normal form . *see* CNF
conversion
  standard 23, 131, 141
context functions 18, 43
counter 59
cross-multiply **131**
CTL **43**
CTL model checking 46
CTL* 43, 46, 64

dual 40, 42, 46, 52

equisatisfiable **24**, **77**, 88, 92
eventuality 40
existential model checking problem **45**, 49

fair Kripke structure **46**
fair path 47
fairness 46, 64
finite path 50
fixed point . . . . . *see* fixpoint
fixed points . . . . . *see* fixpoints
fixpoint **44**
fv 76

GBA . . . . . *see* generalised Büchi automaton
generalised Büchi automaton . . 36
GPVW 178
grasp 10
greatest fixpoint **44**

hole 18, 44

inedge **130**

$K(\hat{M})$ 37
k-l-loop **55**
k-loop **55**
k-loop path 54–57
k-prefix 50
k-prefix path 50
Kripke structure **36**
  fair **46**
  symbolic **36**

least fixpoint **45**
left **32**, 130
linear formula 130
linear temporal logic . . . . . *see* LTL
  with past . . . . . *see* PLTL
linear tree **130**
lit 21
literal . . . . . *see* CNF, literal
$\mathsf{loop}_{k,l}$ 55
loopback point **55**
LTL 38–41, 44
ltl 38

MetateM 72
model checking 45
  bounded . . *see* bounded model checking
  CTL *see* CTL model checking
  LTL . *see* LTL model checking
  symbolic . *see* symbolic model checking
modulo-*n* counter . . . . . *see* counter
monotonicity 27, 79, 83

negation normal form . . . *see* NNF
NNF **22**, 40, 131
Nnf 23, 33, 41
nnf 22
Nonnengart, Rock and Wiedenbach 140, 154
normal form
  clause . . . . . *see* CNF
  conjunctive . . . . . *see* CNF
  negation . . . . . *see* NNF
  separated . . . . . *see* SNF
NP 4

NP-complete 4

op **32**

path 37
  k-l-loop . . . . . *see k*-loop path
  k-loop . . . . . *see k*-loop path
  k-prefix . . . . . *see k*-prefix path
  finite . . . . . *see* finite path
  infinite . . . . . *see* infinite path
  Kripke structure 37
period **55**
π xvii
ϖ . . . . . xvii, **50**, 50n
ϱ 105n
Plaisted and Greenbaum 26, 28, 153
planning 4
PLTL **42**
pltl 42
pol **131**
polarity **27**, **131**
prop 16
propositional logic 16
  with quantification over time *see* QTPL
PSNF 94

qltl 76
QTPL 39

$\mathbf{R}_{\sqsubset V}$ 143
$\mathbf{R}_{\sqsupset V}$ 143
RBC **31**, 30–155
  conversion to CNF . . . 33, 129
rbc 134
reduced Boolean circuit . *see* RBC
renaming **25**, 81
ρ xvii
ϱ xvii
right **32**, 130

SAT 19
satisfiability 19
  equisatisfiable **24**
  partial solution **20**
  problem **19**
  satisfiable **19, 20**
  satisfying assignment **19**

solution **19**
solver 19
unsatisfiable **19, 20**
semantic equivalence **17**
semantics
complete . . . . . *see* complete semantics
LTL 40
k-loop 58
k-prefix 54
PLTL 43
propositional logic **16**
QLTL
k-loop 109
k-prefix 106
denotational 78
sound . . . . *see* sound semantics
separated normal form . . . *see* SNF, 71–126
of CTL 73
of LTL 73
of PTL **73**, 71–73
rule **73**
sib **130**
sign **32**, 133
snf 73
$\mathsf{snf}_{\mathsf{L}}$ 75
snfrule 73
$\mathsf{snfrule}_{\mathsf{L}}$ 74
sound semantics 51
source **130**
SPIN 1, 177, 184
start operator **74**
substitution **18**, **43**
superformula 143
symbolic model checking . . 1–2, 7, 10, 38, **46**, 155, 205
syntactic equivalence **17**

target **32**, 130
Tarski-Knaster 83
temporal logic 37
transition relation 36
tree RBC **130**

universal model checking problem **45**

var **32**, 133
VWAA 188

weak until 40